Amrita Vishwa Vidyapeetham

TIFAC-CORE in Cyber Security

20CYS312 - Principles of Programming Languages

Assignment-01: Exploring Programming Paradigms

BASI REDDY ROHITH REDDY

21st January, 2024

Paradigm 1: Object Oriented Programming(OOP)

Object Oriented Programming(OOPS) is a paradigm that focuses on "objects" as the name itself says. We use classes and objects to implement OOPS.

Objects: Objects are nothing but real or abstract items that contain data to define the object and methods that can manipulate that information. Thus the object is a combination of data and methods.

Classes: Class is a group of objects that has the same properties and behavior and the same kind of relationship and semantics. Once a class has been defined, we can create any number of objects belonging to thy class. Objects are variables of the class. Each object is associated with data of the type class with which they are created; this class is the collection of objects of a similar type.

There are four main features of OOPS i.e..,

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

1. Encapsulation

Encapsulation is the process of wrapping data into a single unit and making different units to be accessed differently.Encapsulation provides an extra layer of security and helps in hiding private information of the class from outside functions.

There are different access control modifiers the common ones being private, public.

1. Private: The private data can only be accessed by the functions(methods) of the same class.

2. Public: The public data can be accessed by any function(method).

2. Inheritance
Inheritance is the property of extending behaviour, methods and variables of one class to another class. There are different types of inheritance:

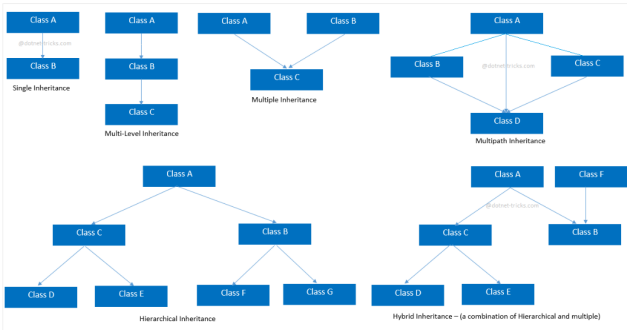

Figure 1: This pic shows us the different types of inheritances.

a. Single inheritance
In this inheritance, a derived class is created from a single base class.

b. Multilevel inheritance
In this inheritance, a derived class is created from another derived class.

c. Multiple inheritance
In this inheritance, a derived class is created from more than one base class.

d. Multipath inheritance
In this inheritance, a derived class is created from other derived classes and also the same base class of other derived classes.

e. Hierarchical inheritance
In this inheritance, more than one derived class is created from a single base class and further child classes act as parent classes for more than one child class.

f. Hybrid inheritance
This is a combination of more than one inheritance.

3. Polymorphism
Polymorphism is the key power of object-oriented programming. It is so important that languages that don't support polymorphism cannot advertise themselves as Object-Oriented languages. Languages that possess classes but have no ability of polymorphism are called object-based languages. Thus it is very vital for an object-oriented programming language. It is the ability of an object or reference to take many forms in different instances. It implements the concept of function overloading, function overriding and virtual functions.

Polymorphism can be of two types:

1. Compile Time(static) Polymorphism
2. Run Time(dynamic) Polymorphism

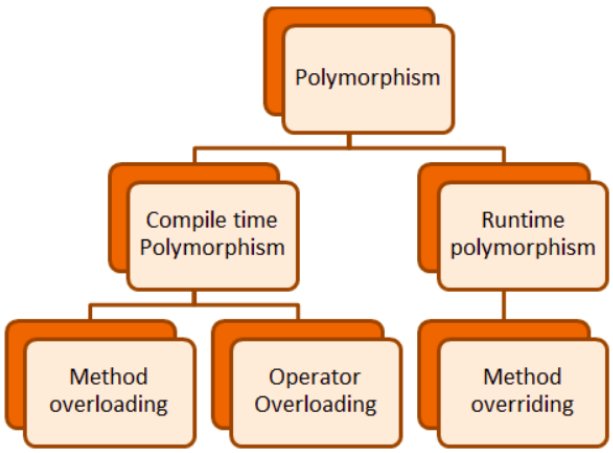

Figure 2: This pic shows us the different types of polymorphism.

1. Compile Time(static) Polymorphism:
It occurs when method overloading or operator overloading occurs.
Method Overloading:
Method overloading involves defining multiple methods in the same class with the same name but different parameter lists.

Operator Overloading: Operator overloading involves defining multiple behaviors for an operator depending on the types of operands.

2. Run Time(dynamic) Polymorphism"
This occurs when we method override.

Method Overriding:
Method overriding involves providing a specific implementation of a method in a subclass that is already defined in its superclass.

4. Abstraction Abstraction means to show only the essential features and hiding unnecessary data. In classes user don't need to know the full data and all methods in a class. For example in a class of lets say bank, the user should not be able to access unnecessary information. OOPs allow abstraction of data and so a programmer can develop the code without having to worry about unnecessary methods.

Language for Paradigm 1: Swift

As similar to OOPs concept, swift allows classes and object creation. But it also has its own functions, methods and access specifiers.

Swift provides quite many handy access control levels which are really helpful while writing our code. Some of the access levels provided SWIFT:
Open access and public access — Entities with this access level can be accessed within the module that they are defined as well as outside their module.

Internal access — Entities with this access level can be accessed by any files within the same module but not outside of it.

File-private access — Entities with this access level can only be accessed within the defining source file.

Private access — Entities with this access level can be accessed only within the defining enclosure.

Now let's see a few examples to understand OOPs in swift better.
Example:

```swift
class ArrayData {

    private let arrayData: [Int]
    private let sortedData: [Int]

    init(array:[Int]) {
        self.arrayData = array
        sortedData = self.arrayData.sorted()
    }

    func sumOfArray() -> Int {
        return self.arrayData.reduce(0, +)
    }

    func sortedArrayData() -> [Int] {
        return sortedData
    }

    func smallestData() -> Int {
        return sortedData.first ?? 0
    }

    func largestData() -> Int {
        return sortedData.last ?? 0
    }
}
```

Figure 3: This code snippet is an example showing a class in swift with encapsulation.

Note: Swift classes use an initialization process called two-phase-initialization to guarantee that all properties are initialized before you use them.

Note: Swift supports single inheritance, multi inheritance and hybrid inheritances. Other types of inheritances can be implemented but not directly.

Example:

```swift
// 1
class Piano: Instrument {
  let hasPedals: Bool
  // 2
  static let whiteKeys = 52
  static let blackKeys = 36

  // 3
  init(brand: String, hasPedals: Bool = false) {
    self.hasPedals = hasPedals
    // 4
    super.init(brand: brand)
  }

  // 5
  override func tune() -> String {
    return "Piano standard tuning for \(brand)."
  }

  override func play(_ music: Music) -> String {
    // 6
    let preparedNotes = super.play(music)
    return "Piano playing \(preparedNotes)"
  }
}
```

Figure 4: This code snippet is an example showing inheritance in swift.

Example:

```swift
func add(_ a: Int, _ b: Int) -> Int {
    return a + b
}

func add(_ a: Double, _ b: Double) -> Double {
    return a + b
}


let sumInt = add(5, 7)
let sumDouble = add(3.5, 2.5)
```

Figure 5: This code snippet is an example showing function overloading in swift.

In this example, we have two functions named add, but they accept different types of parameters (Int and Double). The appropriate function is selected at compile-time based on the types of arguments passed.

Example:

```swift
struct Vector2D {
    var x, y: Double

    static func + (lhs: Vector2D, rhs: Vector2D) -> Vector2D {
        return Vector2D(x: lhs.x + rhs.x, y: lhs.y + rhs.y)
    }
}

let vector1 = Vector2D(x: 1.0, y: 2.0)
let vector2 = Vector2D(x: 3.0, y: 4.0)

let resultVector = vector1 + vector2
```

Figure 6: This code snippet is an example showing operator overloading in swift.

In this example, we have overloaded the + operator for the Vector2D struct, allowing us to perform addition with instances of Vector2D.

Example:

```swift
class Animal {
    func makeSound() {
        print("Some generic animal sound.")
    }
}

class Dog: Animal {
    override func makeSound() {
        print("Woof! Woof!")
    }
}

// Usage
let genericAnimal = Animal()
let dog = Dog()

genericAnimal.makeSound()
dog.makeSound()
```

Figure 7: This code snippet is an example showing method overriding in swift.

In this example, the Dog class inherits from the Animal class and overrides the makeSound method. When we call makeSound on a Dog instance, it uses the overridden implementation in the Dog class.

Paradigm 2: Concurrent Programming

This is an advanced technique allowing simultaneous execution of multiple tasks, improving performance and responsiveness of a program. Tasks are known as threads or processes that run independently, share resources, and interact with each other.

In concurrent programming, the through increases due to parallelism and multi core processing of tasks. The basic principles of this type of programming are:

1. Parallelism
2. Non-determinism
3. Synchronization

The main feature here is parallelism.

Concurrency focuses on managing task dependencies and communication, while parallelism focuses on actual parallel execution of tasks on multiple processing units.

Example:

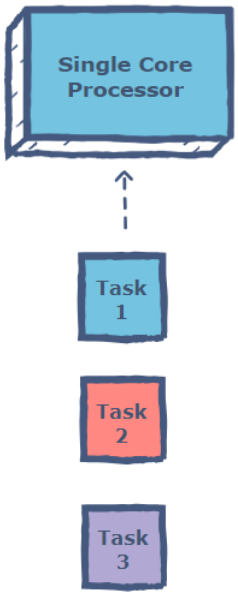

Figure 8: This is an illustration of concurrency.

Example:

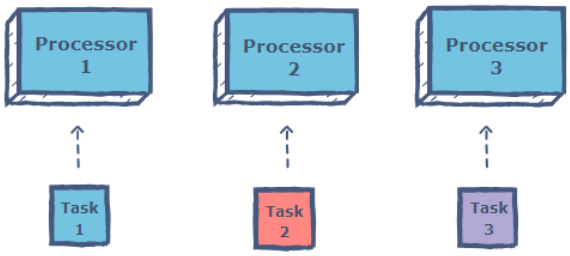

Figure 9: This is an illustration of parallelism.

Concurrency is usually implemented using processes and threads. Processes are separate instances of a program with their own memory space, while threads share the same memory space within a process. We use them to multitask in an application.

Concurrent tasks should be as independent as possible to avoid unnecessary dependencies and enable parallel execution.

Mechanisms like semaphores, mutexes are used to control access to critical section(CS) i.e.., shared resources.

There are a few problems that may arise in this process:

Deadlock occurs when two or more tasks are unable to proceed because each is waiting for the other to release a resource.

Starvation occurs when a task is perpetually denied access to a resource it needs.

Example:

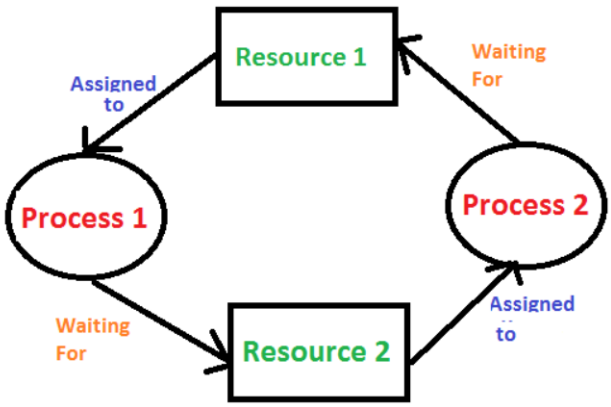

Figure 10: This is an illustration of a deadlock.

The deadlock has the following characteristics:

1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

1. Mutual Exclusion:
   Each resource once allocated to one process can't be shared with other processes until released from that process.(non-sharable)

2. Hold and Wait:
   A process holds at least one resource and is waiting to acquire additional resources that are currently held by other processes.

3. No Preemption:
   Resources cannot be forcibly taken away from a process; they must be released voluntarily.

4. Circular Wait:
   A set of processes P0, P1, ..., Pn exists such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2, ..., Pn-1 is waiting for a resource held by Pn, and Pn is waiting for a resource held by P0.

   Deadlock can be prevented by making any one of the above four conditions invalid also we can use algorithms like **Banker's Algorithm** to avoid deadlock.

It is important to communicate between concurrently executing tasks. This allows them to share information and synchronize their actions. Using message queues or channels to pass data between different components of a distributed system.

Asynchronous Programming: Asynchronous programming(not in sync) enables tasks to proceed without waiting for other tasks. Tasks can execute out of order, and asynchronous programming is often associated with non-blocking I/O operations. We use asynchronous callbacks for handling user inputs or external events in this.

Language for Paradigm 2: Go

Go is well-suited for Concurrency because of its lightweight Goroutines and built-in Channel type. Goroutines are lightweight threads that can be created easily and have low overhead, allowing for the efficient creation of thousands or even millions of concurrent processes. Channels are built-in data structures that facilitate communication between Goroutines, enabling safe and efficient synchronization of data access. It also has WaitGroups to keep the program on wait until other parallel processes are done.

Goroutines: A Goroutine is an independent function in go that executes simultaneously in some separate lightweight(not much demanding) threads.

WaitGroups: You can use WaitGroups to wait for multiple goroutines to finish. A WaitGroup blocks the execution of a function until its internal counter becomes 0.

Channels: In concurrent programming, "Go" provides channels that you can use for bidirectional communication between Goroutines. These channels are one of the more convenient ways to send and receive notifications.

Example:

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    go helloworld()
    time.Sleep(1 * time.Second)
    goodbye()
}

func helloworld() {
    fmt.Println("Hello World!")
}

func goodbye() {
    fmt.Println("Good Bye!")
}
```

Figure 11: This code is an example showing a simple go concurrency program.

Note: Here we are using Sleep() to wait for the thread to complete its execution before proceeding. In "Go" we have WaitGroups functions like:
1.wg.Add(int): This method indicates the number of goroutines to wait. It is used to say how many number of Goroutines need to be executed(initialize internal counter).
2.wg.Wait(): This method blocks the execution of code until the internal counter becomes 0.
3.wg.Done(): This will reduce the internal counter value by 1.

Now let's see an example code with the above mentioned WaitGroups.

Example:

```go
package main

import (
    "fmt"
    "sync"
)

func main() {
    var wg sync.WaitGroup
    wg.Add(2)
    go helloworld(&wg)
    go goodbye(&wg)
    wg.Wait()
}

func helloworld(wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println("Hello World!")
}

func goodbye(wg *sync.WaitGroup) {
    defer wg.Done()
    fmt.Println("Good Bye!")
}
```

Figure 12: This code is an example showing WaitGroups in concurrent go program.

Channels allow us to send messages between goroutines. But we should also close the channel after completing its usage if we do not want it to be used by any goroutine again.

Go has a special statement called select which works like a switch but for channels:
The select statement is often used to implement a timeout:
Example:

```go
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
}
```

Figure 13: This code snippet is an example showing select with time.After in concurrent go program.

time.After creates a channel and after the given duration will send the current time on it.

Example:

```
func main() {
  c1 := make(chan string)
  c2 := make(chan string)

  go func() {
    for {
      c1 <- "from 1"
      time.Sleep(time.Second * 2)
    }
  }()

  go func() {
    for {
      c2 <- "from 2"
      time.Sleep(time.Second * 3)
    }
  }()

  go func() {
    for {
      select {
      case msg1 := <- c1:
        fmt.Println(msg1)
      case msg2 := <- c2:
        fmt.Println(msg2)
      }
    }
  }()

  var input string
  fmt.Scanln(&input)
}
```

Figure 14: This code snippet is an example showing select with different messages depending on time

This program prints "from 1" every 2 seconds and "from 2" every 3 seconds. select picks the first channel that is ready and receives from it (or sends to it). If more than one of the channels are ready then it randomly picks which one to receive from. If none of the channels are ready, the statement blocks until one becomes available.

Example:

```
select {
case msg1 := <- c1:
  fmt.Println("Message 1", msg1)
case msg2 := <- c2:
  fmt.Println("Message 2", msg2)
case <- time.After(time.Second):
  fmt.Println("timeout")
default:
  fmt.Println("nothing ready")
}
```

Figure 15: This code snippet is an example showing select with different messages depending on time

The default case arises when no channel is ready.

Closing the channel:
Closing the channel indicates that no more values should be sent on it. We want to show that the work has been completed and there is no need to keep a channel open.

In the below code we use channel to send across a message as greeting from one goroutine to another and then close the channel at the end. We can use such messages to synchronize the progress of the goroutines.
Example:

```go
package main

import (
    "fmt"
    "time"
)

func main() {
    msg := make(chan string)
    go greet(msg)

    time.Sleep(2 * time.Second)

    greeting := <-msg

    time.Sleep(2 * time.Second)
    fmt.Println("Greeting received")
    fmt.Println(greeting)

    _, ok := <-msg
    if ok {
        fmt.Println("Channel is open!")
    } else {
        fmt.Println("Channel is closed!")
    }
}

func greet(ch chan string) {
    fmt.Println("Greeter waiting to send greeting!")

    ch <- "Hello Rwitesh"
    close(ch)

    fmt.Println("Greeter completed")
}
```

Figure 16: This code is an example showing message interaction between goroutines in go.

Analysis

Swift features

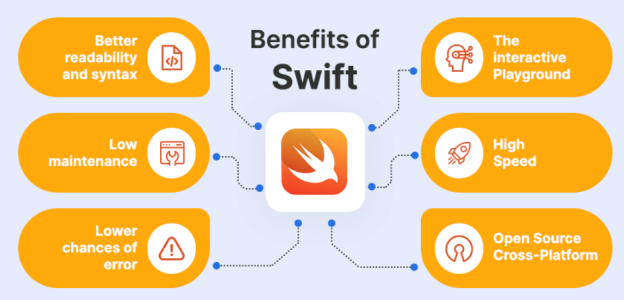

Figure 17: This picture shows some of the benefits of swift language.

Some of the features that swift language has include:
Powerful generics: Generics allow you to write flexible, reusable functions and types that can work with any type.

Structs and classes: Swift allows you to define a structure or class in a single file, and the external interface is made available for other code to use.

Cross-platform: Swift supports all Apple platforms, Linux, Windows, and Ubuntu.

Automatic Reference Counting (ARC): Determines which instances are no longer in use and automatically removes them.

Accessible: Swift is a widely available, free programming language. As an open-source language, you'll find third-party tools, help when you need it, and a knowledgeable community of like-minded users to help you learn Swift.

Advantages of using OOP in swift

1. Ease of Use and Readability: Swift's syntax is clean and expressive and it also has classes and objects in it so we can easily use OOP. It is easy to manage and helps in code organization.

2. Code Reusability: Instead of rewriting classes which have some common methods, we can use inheritance and polymorphism to allow code reuse in swift. Subclasses inherit behaviour from a superclass which reduces redundancy. And polymorphism allows different behaviour for different states which reduces our need to rewrite different methods with different names.

3. Encapsulation: Swift has access control mechanisms (private, internal, public) which help in encapsulation, restricting access to certain parts of the code. We can use these access controls to create robust and secure applications.

4. Support for Abstraction: Swift supports abstraction through protocols and interfaces. This allows developers to define abstract types and implement them in a concrete manner, promoting flexibility in design.

Drawbacks of using OOPs

These features can make the code harder to understand, debug, and test, and can introduce errors and bugs that are difficult to detect and fix.

Another drawback of OOP is that it can consume more memory and CPU resources than other paradigms, such as procedural or functional programming.

Real world applications of OOPs:

1. Software Development:

OOPs, are widely used in software development because it allows developers to design modular and reusable code. It improves code organization by encapsulating data and methods within objects, making complex software systems easier to manage and maintain. Furthermore, OOPs promotes code reuse through concepts such as inheritance and composition, allowing developers to build on existing codebases and eliminate repetition.

2. Development of Graphical User Interfaces (GUIs):

OOPs lends itself well to GUI development, giving a natural approach to model and representing graphical elements. Developers may construct engaging and intuitive user interfaces using classes and objects. OOPs, and frameworks such as Java's Swing and C's Windows Presentation Foundation (WPF) provide rich libraries and tools to help developers create visually beautiful and user-friendly programs.

3. Game Development:

OOPs are commonly used in game development due to their ability to express game things as objects. Complex interactions between people, objects, and settings are common in games. Developers can use OOPs to describe these entities as objects and define their behaviors and characteristics. OOPs ideas are used by game engines such as Unity and Unreal Engine, allowing developers to build immersive and engaging gaming experiences.

4. Database Management Systems:

Encapsulation and abstraction are OOPs ideas used in database management systems (DBMS). OOPs languages like Python and Java have libraries like JDBC and SQLAlchemy that ease database interaction. Database activities can be encapsulated within objects, enhancing code organization and maintainability. OOPs also enables the building of data models, which makes mapping database tables easier.

5. Mobile App Development:

OOPs are widely utilized in mobile app development, powering popular platforms such as Android and iOS. Languages that adhere to OOPs concepts, like Java and Swift, enable developers to create robust and scalable mobile applications. OOPs allows for code reuse, making designing programs for multiple platforms easier while sharing similar code logic.

Swift is commonly used app development in iOS, macOS, Linux, where OOP principles are extensively employed. UIKit and SwiftUI frameworks heavily rely on OOP for building user interfaces, managing data models, and handling event-driven programming.

Server-side Swift frameworks like Vapor and Kitura utilize OOP to structure server-side applications, define models, and handle routing.

Go features

Figure 18: This picture shows some of the features of go.

Arguably Go's most famous feature, concurrency allows processing to be run in parallel over the number of available cores on the machine or server. Concurrency makes most sense when separate processes do not rely on each other (do not need to run sequentially) and where time performance is critical.

The 'go' keyword before a function invocation will run that function concurrently.

In Go, when you supply a primitive (number, boolean or string) or a struct (the rough equivalent of a class object) as a parameter to a function, Go always makes a copy of the value of the variable.

In many other languages such as Java, Python and JavaScript, primitives are passed by value, but objects (class instances) are passed by reference, meaning that the receiving function actually receives a pointer to the original object, not a copy thereof.

This means that any changes made to the object in the receiving function are reflected in original object.

In Go, structs and primitives are by default passed by value, with the option of passing a pointer , through the use of the asterisk operator.

Advantages of using concurrency in go

Here are the main advantages of concurrency:

It allows multiple applications to be executed or run simultaneously, thus increasing efficiency and the total output of workstations.

It promotes better utilisation of resources and allows unused assets or data to be accessed by other applications in an organised manner. This also improves average response time by reducing the waiting time between threads. Fundamentally, applications do not need to wait for the active operation to finish or other applications to complete their functions. This is done by using free resources when given the opportunity as threads can use a variety of resources to complete their objective.

It also helps in improving the operating system's performance. This is possible due to different hardware resources being accessed simultaneously by separate applications or threads. This is different from the point above as it helps with simultaneous use of the same resources as well as the parallel use of different resources. It can also help integrate different resources or applications seamlessly to finish the main objective as fast as possible.

Drawbacks of using concurrency

Parallelly running applications must be coordinated, synchronised, and scheduled in a highly organised manner with special importance placed on allocation and sequential order.

Concurrently running applications must be protected from one another to cause as little interference as possible.

Performance can even be negatively affected or degraded due to too many processes being executed at the same time.

Advantages of concurrent programming in go:

1. Concurrent Web Scraping

Web scraping is the process of extracting data from websites. With Go's concurrency features, we can scrape multiple websites concurrently, significantly improving the scraping speed. The code sample provided showcases concurrent web scraping using goroutines and a wait group. Each website is scraped concurrently, enabling us to fetch data from multiple sources simultaneously.

2. Parallel Image Processing

Image processing tasks, such as resizing or applying filters, can be computationally intensive. Go's concurrency capabilities allow us to process multiple images in parallel, significantly reducing the overall processing time. The code sample demonstrates how to leverage goroutines and a wait group to process a collection of images concurrently. Each image is loaded, processed, and saved concurrently, maximizing efficiency.

3. Concurrent File Downloads

Downloading files from remote servers can be time-consuming, especially when dealing with large files or multiple files. Go's concurrency features enable us to download files concurrently, taking advantage of available network bandwidth and reducing the download time. The code sample illustrates how to download multiple files concurrently using goroutines and a wait group. Each file is downloaded concurrently, allowing for faster retrieval of the desired content.

4. Real-Time Data Processing

Real-time data processing involves handling incoming data streams and performing operations on them as they arrive. Go's concurrency capabilities make it well-suited for real-time data processing tasks. The code sample demonstrates how to process a stream of data concurrently using goroutines and a wait group. Each data point is processed independently, allowing for real-time analysis or computations.

5. Concurrent Network Servers

Building network servers that can handle multiple client connections concurrently is a common requirement. Go's concurrency features enable us to create efficient and scalable network servers. The code sample showcases a concurrent network server implemented using goroutines. Incoming client connections are accepted concurrently, and each connection is handled independently, allowing for concurrent communication with multiple clients.

6. Concurrent Web Server

Creating a concurrent web server is another powerful use case for Go's concurrency capabilities. A concurrent web server can handle multiple incoming HTTP requests concurrently, maximizing throughput and responsiveness.

Drawbacks of concurrent programming in go

1. Lack of Native Thread Control:
   Go routines do not provide fine-grained control over operating system threads, unlike Java and C. This limitation may be a drawback in certain scenarios that require explicit thread management or low-level control over system resources.

2. Limited Synchronization Primitives:
   While channels simplify synchronization in many cases, Go provides limited low-level synchronization primitives compared to Java and C. This may make it more challenging to implement complex synchronization patterns or advanced concurrent algorithms.

Comparison

OOPs and concurrent paradigms doesn't have much in common. Some object oriented languages provide concurrency features also.

OOPs concentrates more on providing encapsulation, abstraction, polymorphism, code reusability, inheritance. It makes the code more readable and simplifies the coding process.

Whereas concurrency provides more throughput through the usage of threads to parallely complete different tasks at the same time. It focuses more on providing sync between parallel tasks and communication between the tasks. It is used to reduce time taken to complete the tasks.

Figure 19: Swift logo.

Figure 20: golang logo.

swift vs go

Swift is developed by apple and is used for software and app development in iOS, macOS, watchOS, and tvOS.

Whereas go is developed by google and is majorly used on server-side applications, in distributed systems and in network programming.

Both languages provide concurrency programming features and have their own properties. Also both languages are simple and easy to learn.

Swift uses Automatic Reference Counting (ARC) for memory management. Swift automatically tracks and manages references to objects, handling memory deallocation.

Go uses automatic garbage collection for memory management. Developers don't need to explicitly manage memory allocation and deallocation in go.

Figure 21: Which is better?.

Challenges Faced

The paradigms had more repeating matter and so i had to filter the best out of the ones I had collected. I did this by taking matter from different sources and taking the best of the lines and also writing my own points in places if needed.

Conclusion

In conclusion, Swift and Go are languages made with different aspects in mind.

Usage of swift for OOPs results in modularity, code reusability. Swift is used mainly for iOS, macOS, watchOS, tvOS app development within the Apple ecosystem.

It's notable features include optionals, Automatic Reference Counting (ARC), Protocol-Oriented Programming.

It is ideal for building modular and maintainable applications within the Apple ecosystem.

Go focuses more on concurrency , simplicity, efficiency. It is well-suited for scalable server-side applications, distributed systems, and cloud services.

Go has its own unique features like Goroutines, Channels, Concurrency Patterns.

It is excellent for handling concurrent tasks efficiently, emphasizing simplicity and scalability.

I reached a conclusion that each language has its own plus points and depending on our need we can prefer one language over the other. The paradigms have not much similarities and doesn't really affect each other. Depending on our project we need to work on, we tend to the paradigm we need.

If the application needs to be more robust and the code to be more readable, scalable and reusable, then we can opt to OOPs in swift else if the website needs to focus more on execution time and needs to complete many tasks in least time, then we may opt to concurrency in Go.

References

https://www.kodeco.com/599-object-oriented-programming-in-swift

https://www.educba.com/object-oriented-programming-paradigm/

https://www.freecodecamp.org/news/concurrent-programming-in-go

https://levelup.gitconnected.com/exploring-the-power-of-concurrency-in-go-through-real-world-examples-25ba691ae4e0

https://mobisoftinfotech.com/resources/blog/ios-development-with-swift-apples-programming-language-of-the-future/

https://www.educative.io/blog/swift-programming

https://www.golang-book.com/books/intro/10

https://futurice.com/blog/gocurrency

https://oxylabs.io/blog/concurrency-vs-parallelism

CONCURRENCY AND OBJECT-ORIENTED PROGRAMMING - Michael L. Nelson, Major, USAF Department of Computer Science, Naval Postgraduate School Monterey, CA 93943