20CYS312 - Principles of Programming Languages
Exploring Programming Paradigms

Assignment-01

Presented by Basi Reddy Rohith Reddy
CB.EN.U4CYS21013
TIFAC-CORE in Cyber Security
Amrita Vishwa Vidyapeetham, Coimbatore Campus

Feb 2024

AMRITA
VISHWA VIDYAPEETHAM

Outline

Object-Oriented

Object-Oriented Programming (OOP) is a programming paradigm that revolves around the concept of "objects." Objects are instances of classes, which are user-defined data types that encapsulate data and the operations that can be performed on that data. OOP is based on four main principles: encapsulation, inheritance, polymorphism, and abstraction.

To know about the principles we first need to know about the basic components(classes and objects).

Class: A blueprint or a template for creating objects. It defines the properties and behaviors that the objects of the class should have. Once a class has been defined, we can create any number of objects belonging to thy class.
Object: An instance of a class that has states and behavior.

Encapsulation:
Encapsulation is the process of wrapping up data and functions into a single unit. It is the most striking feature of the class. It helps in hiding the internal details of an object and exposing only what is necessary.

Inheritance:
Inheritance is the property whereby one class extends another class's properties, including additional methods and variables. The original class is called a superclass, and the class that exceeds the properties are called a subclass. It promotes code reusability and establishes a relationship between classes.

Polymorphism:
In geek terms, polymorphism means the ability to take more than one form.
Polymorphism allows objects of different classes to be treated as objects of a common superclass. It enables flexibility in code by allowing methods to be used interchangeably.(exhibit different behavior in a different instance)

Abstraction:
Abstraction involves hiding unnecessary details. It does not represent the background details and explanation. It allows programmers to focus on what an object does rather than how it achieves its functionality.(We dont need to know how a method works fully to implement it and neither does our code needs that information)

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: 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.

Swift supports single inheritance, meaning a class can inherit from only one superclass.
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: This code snippet is an example showing inheritance in swift.

Swift allows for method overriding enabling polymorphism. Subclasses can provide their own implementation of methods defined in the superclass.

Example:

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

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

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

Real World Applications:

App Development: 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.

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

Concurrent

The Concurrent Programming paradigm is the idea of managing multiple tasks or processes simultaneously,(at the same time) enabling more efficient use of resources and improved responsiveness in computing systems.
Here are five main features of the Concurrent Programming paradigm:

1. Parallel Execution: Concurrent Programming often involves parallel execution, allowing multiple tasks to be performed simultaneously. As the name says it uses multiple processors to achieve more throughput by running computations on different cores and thus reducing execution time.

2. Processes and Threads: 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.

3. Synchronization and Coordination: Concurrent tasks often need to synchronize their progress to avoid conflicts and ensure data consistency. To do that locks, mutex, semaphores are used for synchronization. We also control access to shared resources, like a critical section, to prevent data corruption.

4. Message Passing and Communication: 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.

5. 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.

Concurrent - Go

In "go" there are Goroutines, channels, waitgroups that we use to implement concurrency.

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: 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: This code is an example showing WaitGrops 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.

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

Real world applications of 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.

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.

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.

Features 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.

Features of concuurent programming in Go

1. Goroutines and Concurrency Control:

Go's goroutines provide a lightweight concurrency model, allowing users to easily create concurrent programs. The language also includes built-in support for synchronization mechanisms like channels and mutexes.

2. Efficient Parallelism:

Go's concurrency model is good to implement parallelism, making it efficient for executing tasks concurrently on multiple cores or processors.

3. Communication with Channels:

Go's channels simplify communication between concurrent tasks, reducing the need for explicit locking. It helps in synchronization of the processes.

4. Scalability and Performance:
Go is designed with scalability in mind, making it suitable for building concurrent and distributed systems. It performs well in scenarios with a high number of concurrent operations and the code readability is also good.

5. Garbage Collection:
Go includes a garbage collector that manages memory automatically. This helps simplify memory management in concurrent programs.

Conclusion:
Comparing both languages and their features, we can see that each language has its own plus points and depending on our need we can prefer one language over the other.

Bibliography

For this presentation I have inferred information from the following sites:
kodeco.com
educba.com
freecodecamp.org
medium
futurice.com

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://futurice.com/blog/gocurrency