Object-Oriented Programming in Go

Encapsulation, Polymorphism, and Abstraction

OOP Concepts in Go

- Go supports OOP principles through struct types and interfaces.
- While Go doesn't have traditional classes, it achieves OOP-like behavior.

Encapsulation in Go

- Encapsulation is achieved by exporting or unexporting fields in a struct.
- Capitalized fields are exported, while uncapitalized ones are not.

```
package shapes

type Rectangle struct {
    Length  float64 // Exported field
    width   float64 // Unexported field
}
```

Encapsulation in C++

- In C++, encapsulation is achieved using access specifiers (`public` , `private` , `protected`) within classes.

```
class Rectangle {
public:
    float Length; // Public field
private:
    float Width; // Private field
};
```

Polymorphism in Go

- Polymorphism is achieved through interfaces in Go.
- Types implicitly implement an interface if they define its methods.

```
type Shape interface {
    Area() float64
}
```

Polymorphism in C++

- In C++, polymorphism is achieved through classes and inheritance.
- Base and derived classes are used for polymorphic behavior.

```
class Shape {
public:
    virtual float Area() const = 0; // Pure virtual function
};

class Rectangle : public Shape {
    // Implement Area() in Rectangle
};
```

Abstraction in Go

- Abstraction is achieved through interfaces in Go.
- Interfaces define a set of methods without specifying the implementation.

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Abstraction in C++

- In C++, abstraction is achieved using abstract base classes and pure virtual functions.

```
class Reader {
public:
    virtual int Read(char* buffer, int size) = 0;
};

class FileReader : public Reader {
    // Implement Read() in FileReader
};
```

Comparison: Encapsulation

- Go uses capitalized field names for exported fields.
- C++ uses access specifiers (public, private, protected) within classes.

```go
type Rectangle struct {
    Length  float64 // Exported field
    Width   float64 // Unexported field
}
```

```cpp
class Rectangle {
public:
    float Length; // Public field
private:
    float Width; // Private field
};
```

Comparison: Polymorphism

- Go uses interfaces to define polymorphic behavior.
- C++ uses classes and inheritance with virtual functions.

```go
type Shape interface {
    Area() float64
}
```

```cpp
class Shape {
public:
    virtual float Area() const = 0; // Pure virtual function
};

class Rectangle : public Shape {
    // Implement Area() in Rectangle
};
```

Comparison: Abstraction

- Go uses interfaces to define abstract behavior.
- C++ uses abstract base classes with pure virtual functions.

```go
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

```cpp
class Reader {
public:
    virtual int Read(char* buffer, int size) = 0;
};

class FileReader : public Reader {
    // Implement Read() in FileReader
};
```

Summary

- Go achieves OOP-like behavior through structs and interfaces.
- Encapsulation, polymorphism, and abstraction are supported.
- C++ uses classes and inheritance for OOP.