Alexander Bauer

Golang vs Java and C++

Golang differs in several ways from C++ and Java. I will begin by listing Golangs features and then compare those to C++ and Java.

- Go has a number of built-in types, including numeric (byte, int32, uint32, etc.), booleans, and strings
- Go has no dynamic castings for arithmetic
- Go has all of the common control flows except try/catch (discussed later)
- Go has no forward declarations, things are declared only once
- Shortens and clarifies code with things like the := construct (foo.Foo * myFoo = new(foo.Foo) is the same as myFoo := new(foo.Foo))
- There is no type hierarchy, types are what they are they don't have to announce their relationships.
- Golang has no type inheritance, interfaces are applied if a type implements the necessary methods without explicitly stating it is implementing an interface, allowing types to easily satisfy many interfaces at once.
- Methods can be implemented for any type
- Interfaces provide a sort of inheritance. Any type that satisfies an interface can be passed as that interface
- Go has a runtime library that implements garbage collection, concurrency, stack management, and more.
- Error handling is done through multi-value returns rather that try-catch-finally idiom
- Golang is statically typed
- Golang deliberately omits : generic programming, assertions, pointer arithmetic, and implicit type conversions
- Golang scope is specified by blocks of code, code within curly braces. Variables, constants, types, functions declared outside of any function are visible across the whole package (the scope is the package block. Inner blocks will use the most recent definition of a variable name.

```
v := "outer"
fmt.Println(v)
{
    v := "inner"
    fmt.Println(v)
    {
        fmt.Println(v)
    }
}
```

```
    {
        fmt.Println(v)
    }
    fmt.Println(v)
```

Outputs

```
        outer
        inner
        inner
        outer
        outer
```

- Golang has its own subtle version of privacy, within a package, if a struct or function has an Uppercase identifier, it will be exported making it public. If a struct or funciton has a lowercase identifier, it will not be exported making it private. This can be tricky because a private struct with an Uppercase field will make that structs field public. Also, everything inside a given package can be seen by the package.
- Go does not have undefined behavior as extensively as C++ does, but there are problems. One is that the exit of a goroutine is not guaranteed to happen before any event in the program. Ex:

```
var a string

func hello() {
    go func() { a = "hello" }()
    print(a)
}
```

does not guarantee something is assigned to a before the print statement since there is no synchronization event

Go differs from C++ in that:

- C++ has manual memory management, which is less safe
- C++ has dynamic castings which can lead to bugs
- C++ has explicit typing and inheritance
- C++ has try and catch blocks
- Go has no pointer arithmetic
- C++ does not support multi-value returns (without direct implementation)

Both languages share OOP and Procedural paradigms

Go differs from Java in that:

- Java has exceptions, Go does not
- Go is much simpler to read than Java
- Go has no generic types (might be added later)
- Java has explicit typing and inheritance
- Java has no undefined behavior

Both have memory safety (although Go has some problems ie data racing)

My Experience

I had a really fun time with this language. Learning the syntax was very simple and the code is very readable. Understanding came fairly quickly. Some things however were trickier, like learning whats private and public and how to include packages, which required some research.

Golang handles OOP through the use of structs, functions, and interfaces. Defining objects is very simple. You create a struct and then create methods for it. Ex:

```
// Field represents a two-dimensional field of cells.
type Field struct {
    s    [][]bool
    w, h int
}

// Set sets the state of the specified cell to the given value.
func (f *Field) Set(x, y int, b bool) {
    f.s[y][x] = b
}
```

One cool feature here is that this method can only be called by a pointer to a field struct. There is no use of -> and . operators like in C++, there is only a dot operator. I feel that the use of pointers is simpler, because they can only be used in stated was. I used a lot of pointer arguments in functions and was not confused at all.

Inheritance is handled through interfaces. I did not write any, but the 2D engine I used did. This interface in fact forced me to learn golang's Inheritance:

```
type Target interface {
    // MakeTriangles generates a specialized copy of the provided Triangles.
    //
    // When calling Draw method on the returned TargetTriangles, the TargetTriangle
    // drawn onto the Target that generated them.
    //
    // Note, that not every Target has to recognize all possible types of Triangles
    // only recognize TrianglesPosition and TrianglesColor and ignore all other prop
    // present) when making new TargetTriangles. This varies from Target to Target.
    MakeTriangles(Triangles) TargetTriangles

    // MakePicture generates a specialized copy of the provided Picture.
    //
    // When calling Draw method on the returned TargetPicture, the TargetPicture wil
    // onto the Target that generated it together with the TargetTriangles supplied
    // method.
    MakePicture(Picture) TargetPicture
}
```

To implement an interface, a data type needs to implement all interface functions, in this case MakeTriangles and MakePicture. I was trying to use PictureData, which did not implement MakePicture as a target so my DrawBackground function was failing. This lead to another nice discovery though. Because of the static typing, I was notified at compilation time that I had to change my code.

If you want to create a "new" object you need to create a function that returns a pointer to a struct. Instead of having member functions, you define functions that structs can use, which ties in to the no forward declaration rule. I like that function declaration and implementation is all in one place. In my program I made a Field and Life struct which I manipulated through functions called on by pointers of the struct.

Things I liked:

- Ease of language on a basic level
- I absolutely love multi-value returns. The ability to return 2 or more things without having to create some type of data structure is so so so convenient
- Ease of access to system through os packages
- Ability to download things from github using go get
- Once I kind of got the hang of interfaces, it was easy to see and use
- The garbage collector
- Inside of a given package, all data is public, which I like because it makes it easier to use your structs. In a complicated project, you could import the package to your main to protect privacy and prevent bad usage. You would only be able to access exported functions and variables. (Upppercase declared)

Things I disliked:

- Lack of libraries and support. I used a 2D game library, that although very nice, was put up less than 2 years ago and I could not find any help on stack exchange or anything. (Thank

god the creator made a wonderful wiki with coding examples). Reading the API was tough at first since I didn't understand interfaces. I have used SDL before and there is tons of support for that with C++ in comparison
- Although I did not run into it, the existence of undefined behavior. Golang has several issues some of which I mentioned above (synchronization and data races)
- Importing a local package as far as I knows requires a specific folder hierarchy, which for a complicated project forces you to stay organized, but for this simple one was a pain
- Not explicitly stating inheritance in a large project could make remembering all the interfaces and implementers difficult, easy solutions would be to add a comment after opening curly bracket stating interfaces the struct will implement

I will definitely use this language again. If it were not for all the API reading I had to do for the 3rd party library I used, this project would have been super easy. And that is because golang is so easy to understand. I want to see if on something a little more complicated, with more packages, it remains as simple.