**CS131 Homework 6: Generics in Go, Java, and OCaml**

Dorian Jimenez
205-376-959
*Discussion 1B*

1. **Java:**
   Looking at Java first, Java does have type inference for generics. Type inference in Java is done by the compiler at compile time.[1] Java does something called Type Erasure however, which is when the type information is discarded at run-time.[2] So when Java uses reflection, the programmer cannot actually see certain information. For example,

   `List<A>` would look the same as `List<B>`

   Java also has wildcards such as

   `List<? extends Number>` or `List<? super Number>`

   In general, type inference in Java is used more or less similarly to Go. Although there are some differences between type inference in Java and Go, I believe the statement applies well to Java.

   As an example, in Java if you have a generic type and Java cannot deduce through type inference what the type of the generic is, you will get an error. The programmer will have to go to the code and provide the necessary type arguments.

   **OCaml:**
   Type inference in OCaml is different from Go. In Go, type inference can fail, and then the programmer would have to provide the necessary type arguments. However, in OCaml, if the compiler is unable to verify the type of a variable, it will make a type variable, `'a`, `'b`, etc. An example of this is:

```
let test = function
    | [] -> 1
    | h :: t -> 2

val test : 'a list -> int = <fun>
```

[1] Type Inference (The Java™ Tutorials > Learning the Java Language > Generics (Updated)) (oracle.com)
[2] Type Erasure in Java Explained | Baeldung

In pure OCaml, there is no way to trick the compiler into failing type inference.[3] With this being said, I don't think this statement necessarily applies to OCaml as well as it did to Java. There are many differences in Go type inference and OCaml type inference. If type inference fails in Go, the programmer must fix the issue, however in OCaml the compiler will just give you a type variable.

2. **When using language-defined container types:**
   Type parameters in Go can be used when writing functions that operate on language-defined special container types (such as slices, maps, and channels) in some cases. If the function code doesn't make any assumptions on the types of the elements, type parameters can be used.

   This applies to Java as well, as Java has features where you can declare Maps or Lists of a type parameter. By doing this, you do not have to rewrite functions that are functionally the same but have different types.

   **General purpose data structures:**
   In "When to use Generics", the author says that you can use type parameters for general purpose data structures (which is something like a slice or map, but one that is not built into the language).

   This applies well to Java, as Java has features where you can declare general purpose data structures with a type parameter. For example, BinTree<T>, which creates an object of any type.

   **For type parameters, prefer functions to methods:**
   The main reason to use functions over methods is because it is much simpler to turn a method into a function than it is to add a method to a type.

   This holds in Java as well. It is much simpler to write a generic function instead of a generic method for general purpose data types. This way the programmer doesn't need to create and implement a generic method that isn't needed.

   **Implementing a common method:**
   If an implementation of a method is the same, independent of what the type is, it's best to use a type parameter.

   This also applies to Java, as generics in Java can help programmers write methods that are implemented the same, no matter what type is being used. By doing this, programmers can avoid rewriting the same code and instead just write the implementation once.

[3] Profiling · OCaml Tutorials

**Don't replace interface types with type parameters:**

In "When to use Generics", the main arguments for why to not replace interface types with type parameters was that omitting the type parameter:

- makes the function easier to read
- makes the function easier to write
- and execution time will likely stay the same

With this in mind, let's compare this to Java. Interfaces in Java are completely abstract, meaning that to access the interface methods, the interface must be implemented first. This is good if you have different implementations of methods, and you want each method to be implemented. Java has type parameters as well, which are identifiers that specify a generic type name.[4] It is useful to use type parameters if you have a function that is implemented the same, but uses different types. Since these are two different use cases, this statement applies equally well to Java.

**Don't use type parameters if method implementations differ:**

In Go, it is recommended to use a type parameter if the implementation of a method is the same for all types, and use an interface type if the implementation is different for each type.

This also holds in Java. If method implementations differ, it is best to use interfaces, which are "fully abstract". This way, but using interface types, you can add different method implementations for different types. With type parameters, this really only works if the implementation of each method for each type is the same. If the process is the same regardless of type, it is best to use type parameters. Since you are using the same function but just changing the type, this would not work if you actually want different implementations for each type.

**Use reflection where appropriate:**

In "When to use Generics", the author suggests to use Reflection if:

- The operation has to support types that don't have methods
  - Interface types won't work
- The operation is different for each type
  - Type parameters won't work

In Java, reflection allows us to "inspect and manipulate classes, interfaces, constructors, methods, and fields at run time".[5] This is similar to Go as Go reflection is a way to manipulate objects with arbitrary types at run-time.

[4] Java - Generics (tutorialspoint.com)
[5] Java Reflection (With Examples) (programiz.com)