Callables: C++ vs Java

Freitag, 4. April 2025 18:52

Author: Damir Ljubic
e-mail: damirlj@yahoo.com

Intro

When I've started with Java - which is very similar with C++/C#, the first obstacle that I had was how to implement the ***higher-order function***: a function that either takes another function as argument, and/or even return one.
Let see how we can accomplish this task in both languages, and what parallels can be drawn.

C++ approach

In C++ this is quite straight forward (well, if you come from C++): you just need to specify the *signature* that function has to satisfy

```cpp
template <typename R>
using callable_type = R (*)(std::string);

void func(std::vector<std::string> input, callable_type<void> callback) {
    for (const auto& in : input) {
        callback(in);
        // std::invoke(callback, in);
    }
}
```

Where the *callable_type* defines the signature of the **free function**.

If we want to restrict to the *non-static member function* of a particular UDT, we can redefine it as

```cpp
template <typename R, class T>
using callable_type = R (T::*)(std::string);
```

@note ***std::function*** is universal, polymorphic placeholder for any callable: but this is out of the scope right now.

Our higher-order function can lift this - visiting all elements in array and applying the same functionality.
This way, our callable type becomes the *customization poin*t (Strategy Design Pattern): any callable that satisfies the signature, can be applied.

Back to signature. It becomes even more obvious using the *std::invoke* utility function, that the instance of the UDT needs to be the *very first argument* of any non-static member function invocation.
Welcome to the world of OO programming.

```cpp
template <class T>
void func(std::vector<std::string> input, callable_type<void> callback, T& obj) {
    for (const auto& in : input) {
        (obj.*callback)(in);
        // (ptr->*callback)(in); // in case that we pass the pointer
        // std::invoke(callback, obj, in);
   }
}
```

Variadic arguments pack

Where C++ prevails is that with C++ we can specify really generic: universal function signature, with arbitrary number of arguments - even of a different type, using **variadic arguments** pack

```
template <typename R, typename...Args>
using universal_callback_type = R (*)(Args&&...);
```

@note We can also add the *const qualifier* to signature - to make the function's enclosing type T immutable

```
template <typename R, typename T, typename...Args>
using universal_callback_type = R (T::*)(const Args&&...) const;
```

@note Actually, we can add *volatile* qualifier as well - which means, that the function will be called on the volatile instance of the enclosing class T

Exception in signature

In C++ - one can also specify explicitly - as part of the function signature, whether the function may throw

```
template <typename R>
using callback_type = R (*)(void) throw (std::logic_error);
```

Starting with C++11 - this is considered deprecated.
Instead - assuming that every function (except destructor) can implicitly throw, as a hint to compiler there is a new operator ***noexcept*** that indicates whether the function may throw - or not (noexcept == noexcept(true)): it can be conditionally expressed

```
template <typename R, typename Arg>
using callback_type = R (*)(Arg) noexcept (std::is_nothrow_copy_constructible_v<Arg>);
```

As a consequence - there will be no stack unwinding in order to propagate the exception to the outer functions that presumably catch and handle exception: but rather if the exception is thrown - the program will terminate (std::terminate)

Java approach

So, how we can accomplish the same with Java?
And Java is indeed the pure OO language.

In Java, we can specify a custom **callback interface**

```
@FunctionalInterface
interface CallbableType<R, T> {
    R apply(T obj);
}
```

This would be equivalent to defining the non-static member function of T, that is **parameterless** - since the very first argument must be the instance on which the method will be invoked.

Then, we define the higher-order function as

```
<R, T> R func(@NonNull CallableType<R, T> callback, T obj) {
    // do something
    return callback.apply(obj);
}
```

This is similar calling the std::invoke, providing the instance of T, as a first argument

We can, on the place where *callback* is expected, provide:

- Reference to the non-static member function of the enclosing class

```
this::<function>
```

- Reference to the non-static member function of another class, for which we need to provide argument as well
  `<Class>::<function>`
- We can provide the **lambda** object on the fly, that satisfies the signature

```
(obj)-> {
    // do something
    return obj.<func>();
}
```

As matter of fact - there are already predefined Functional Interface which are part of the **java.util.function** package, that is introduced with Java 8 SDK.
Function<R, T> offers the same *apply*() callback as our manually written interface.
Actually, it's callbacks interface - since it provides the signature for three additional callbacks.
It's even composable, since you can instantiate it with the callable that will be applied first, and then we can specify additional callback, that will be invoked consequently.

@note There are other useful predefined functional interfaces, like *Consumer<T>*==Functional<Void, T> to support the functional programming style in Java

Exception in signature

Interesting enough, the Exception Type can be also part of the function signature in Java

```
@FunctionalInterface
interface CallableType<R, T> {
    R apply(T obj) throws RemoteException;
}
```

Or to be generic as possible

```
@FunctionalInterface
interface CallableType<R, T, E extends Exception> {
    R apply(@NonNull T obj) throws E;
}
```

Our higher-order function may preserve the exception indication in signature

```
<T, R, E extends Exception> R invoke(CallableType<T, R, E> f, T arg) throws E {
    // do something
    return f.apply(arg);
}
```

or it can handle it internally.

In case that due to *interoperability* - the API expects the Function Interface which is not throwable, and we use the one which may throw, we can write the safe wrapper

```
static <T, R> Function<T, R> safeWrap(ThrowingFunction<T, R, ?> f) {
    return t -> {
        try {
                return f.apply(t);
            } catch (RuntimeException e) {
                throw e; // Re-throw unchecked exceptions
            } catch (Exception e) {
                throw new IllegalStateException("Unexpected checked exception", e);
            }
   };
}
```

Variadic arguments pack

Unlike C++ - Java doesn't support variadic arguments pack: at least not for expressing the arbitrary number of heterogenous types.
It supports only variadic arguments list of the same type

```
@FunctionalInterface
interface CallableType<R, T, E extends Exception> {
    R apply(@NonNull T... objs) throws E;
}
```

Conclusion

At the end - comparing these two approaches, Java is doing quite well, considering the fact that the generic - template programming was not originally part of the language core - it's added afterwards, with Java 4 SDK - implementing it as *type erasur*e (stripping all type information - and treating it as Object inside the function template).
https://github.com/damirlj/modern_cpp_tutorials?tab=readme-ov-file#tutorial-7--type-erasure-

The only where Java is inferior in fulfilling this particular task - is the fact that heterogenous variadic argument pack is not supported (nor the fold expressions, type-traits, auto type deduction and all other mechanics of template metaprogramming).