Week 4

Tuesday, April 11, 2023 9:08 AM

Week 4:

- java 8 features (Stream, Lambda, Functional Interfaces)
- Completable Future.
- Fork-Join Framework

Mutex vs Semaphore :
https://stackoverflow.com/questions/4039899/when-should-we-use-mutex-and-when-should-we-use-semaphore

Semaphore :

A semaphore is an integer variable, shared among multiple processes. The main aim of using a semaphore is process synchronization and access control for a common resource in a concurrent environment.

Functional Interface :

https://blog.knoldus.com/functional-interfaces-in-java-8/

Lambda :

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, such as an interface that contains only one method, then the syntax of anonymous classes may seem unwieldy and unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such as what action should be taken when someone clicks a button. Lambda expressions enable you to do this, to treat functionality as method argument, or code as data. So when we have a functional interface, it's just enough if we specify the parameters and body of the function because they're only important.

Lambda provides backward compatibility (new features like lambda are compatible with older versions of code) which many languages don't provide.

Though it looks like a replacement for anonymous inner class. They don't actually replace anonymous inner class i.e. lambda code is not replaced with anonymous inner class internally. Anonymous inner class generate class files while lambda don't generate class files.

Stream API :

In Java, a Stream is a powerful and expressive API introduced in Java 8 that provides a functional programming approach for processing collections of data. It allows for efficient and concise manipulation of data in a declarative manner, without the need for explicit iteration or loops.
Streams in Java are immutable, which means that once a stream is created, its elements and their order cannot be changed.
Any operation performed on a stream creates a new stream, leaving the original stream unchanged.

In Java, Predicate, Supplier, Consumer, and Function are functional interfaces that are part of the java.util.function package, introduced in Java 8 as part of the Java Stream API. These functional interfaces provide a way to represent and pass around behavior in a functional programming style.

1. **Predicate<T>:** It represents a function that takes an input of type T and returns a boolean value. It is often used for filtering elements in a collection based on a condition. It has a single method test(T t) which returns a boolean value.
2. **Supplier<T>:** It represents a function that takes no input and returns a value of type T. It is often used for providing values or generating data on demand. It has a single method get() which returns a value of type T.
3. **Consumer<T>:** It represents a function that takes an input of type T and performs an action on it, but does not return any value. It is often used for performing operations that have side effects, such as printing or logging. It has a single method accept(T t) which takes an input of type T and performs an action on it.
4. **Function<T, R>:** It represents a function that takes an input of type T and returns a value of type R. It is often used for transforming or mapping elements from one type to another. It has a single method apply(T t) which takes an input of type T and returns a value of type R.

Fork Join Pool :

Fork/Join Framework: The Fork/Join framework is a built-in framework in Java that provides support for parallelism by dividing tasks into smaller sub-tasks that can be executed in parallel on multiple CPU cores or processors. It is useful for parallel processing of large tasks that can be divided into smaller independent tasks.

In Java, the ForkJoinPool uses the number of available processors as the default parallelism level, which is determined by Runtime.availableProcessors() method. This means that by default, the commonPool() in ForkJoinPool will create as many threads as the number of available processors in the system.

If a CPU has 4 cores, it can potentially execute up to 4 threads concurrently, assuming each core is available and not busy with other tasks

With ForkJoinPool's constructors, we can create a custom thread pool with a specific level of parallelism, thread factory and exception handler. Here the pool has a parallelism level of 2. This means that pool will use two processor cores.