Finals Notes (CS2030)

“Lazy”

Wrap your intentions in a Supplier

```
Supplier<Integer> supplier = () -> list.stream()
                            .map(x -> processsUrl(x))
                            .reduce(0, (x, y) -> x + y)
```

When you have cyclic dependancy

B dependant on A

A dependant on B

Break the dependancy: Introduce an interface C, where A implements C, B dependant on C

or use Anonymous Inner Class

```
class ImList<T> {
    private final Supplier<List<T>> list;

    ImList() {
        this.list = new ArrayList<T>();
    }

    private ImList(Supplier<List<T>> list) {
        this.list = list;
    }

    ImList<T> add(T t) {
        return new ImList<T>(() -> {
            System.out.println("Adding: " + t);
            List<T> newList = new ArrayList<T>(this.list.get());
            newList.add(t);
            return newList;
        }
    };
}
```

Anonymous Inner Class

Key ideas:

- A static method in an interface can only be called through that interface
- Not be called by any object or an IMPLEMENTATION of the interface
- No difference from creating a concrete class and implementing the methods of the interface

```
interface Foo {
    static Foo of() {
        return new F();
    }
}

class F implements Foo {
}

new Foo() //error, you cannot instantiate an interface
new F().of() //error, static method of interface can only be called by interface
Foo foo = new F()
foo.of() //error, cannot even if called by implementation of the interface
```

Usage:

- Avoid cyclic dependencies by,
  Anonymous classes can be used to instantiate abstract classes, and subsequently, override methods, to create other anonymous classes within them.
- Single Abstract Method interfaces
- Coupled with interfaces that have static methods (e.g: factory methods like of()), anonymous classes can be used to ensure that instances of a class no longer have any static methods that are callable. (this was one of the exercises, Maybe plus)

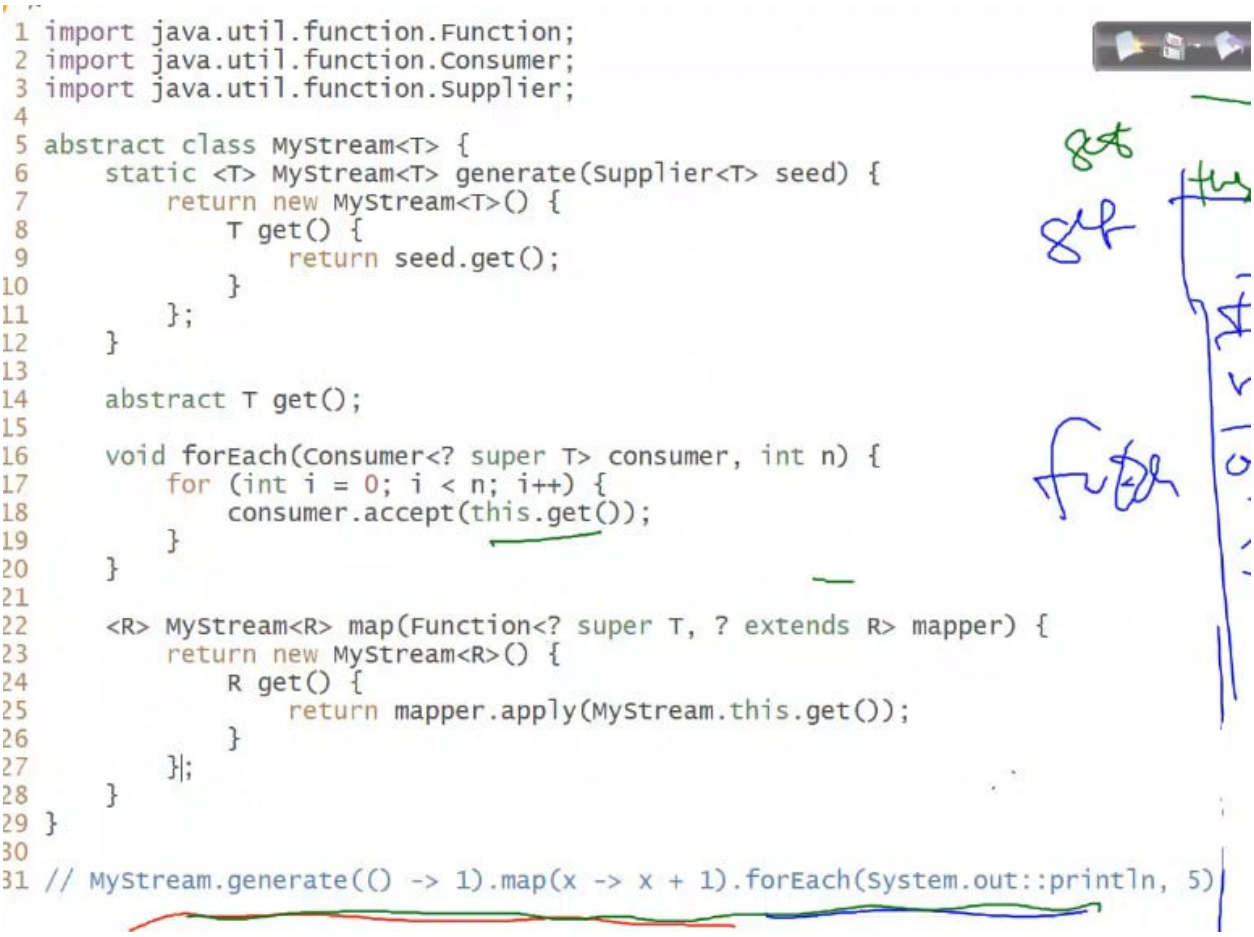
Abstraction

Because Abstract classes/Interfaces are unable to be initialised, the Anonymous Inner Class serves as a Concrete class implementation of the parent Abstract class/Interface

```
interface Maybe<T> {

  static <T> Maybe<T> of(T value) {
    return new Maybe<T>() {
    ...
    }

// the new Maybe<T>() is telling java that you are trying to create a anonymous
// inner class that implements Maybe interface
// returning a new Maybe<T>() implementation as an A.I.C
// creating the class inside and returning 1 instance of the implementation
```



```java
import java.util.function.Function;
import java.util.function.Consumer;
import java.util.function.Supplier;

abstract class MyStream<T> {
    static <T> MyStream<T> generate(Supplier<T> seed) {
        return new MyStream<T>() {
            T get() {
                return seed.get();
            }
        };
    }

    abstract T get();

    void forEach(Consumer<? super T> consumer, int n) {
        for (int i = 0; i < n; i++) {
            consumer.accept(this.get());
        }
    }

    <R> MyStream<R> map(Function<? super T, ? extends R> mapper) {
        return new MyStream<R>() {
            R get() {
                return mapper.apply(MyStream.this.get());
            }
        };
    }
}

// MyStream.generate(() -> 1).map(x -> x + 1).forEach(System.out::println, 5)
```

How A.I.C can be used to return a new Instantiation of an object with a different type

```
interface ImList<E> {

//specify the contracts of the A.I.C
  ImList<E> add(E elem);

  static <E> ImList<E> of() {
      return ImList.of(List.of());

  static <E> ImList<E> of(List<? extends E> list) {
      return new ImList<E>() {
        private final ArrayList<E> elems = new ArrayList<E>(list);

        public ImList<E> add(E elem) {
          List<E> newList = new ArrayList<E>(this.elems);
          newList.add(elem);
          return ImList.of(newList);
        }
  }
}
```

Static methods in interfaces cannot be overriden in implementation classes

Asynchronous Programming

Synchronous

```
void foo(int m, int n) {
  B b = f(new A());
  C c = g(b, m);
  D d = h(b, n);
  E e = n(c, d);
}

/*
Calling foo(5, 10) would result in a total of 20s call time because
g() waits for f(), h() waits for g()
*/
```

Using CompletableFuture to make processes asynchronous

```
E foo(int m, int n) {
  Supplier<B> suppB = () -> f(new A());

  CompletableFuture<B> cfB = CompletableFuture.supplyAsync(suppB);
  CompletableFuture<C> cfC = cfB.thenApply(x -> g(x, m));
  CompletableFuture<D> cfD = cfB.thenApplyAsync(x -> h(x, n));
  CompletableFuture<C> cfE = cfC.thenCombine(cfD, (c,d) -> n(c,d));

  E e = cfE.join();

  return e;
}

/*
Now upon the calling of foo(5,10), it would only take 15s as C and D
are running on different threads (through the use of Async)
*/
```

CompletableFuture<T>

```
//runAsync or supplyAsync to create
//"then"
//followed by Accept, Combine, Compose, Run
//Async(or empty if synchronous)

CompletableFuture<T> cf = CompletableFuture.supplyAsync(() -> f(new A()));

//example
CompletableFuture<Integer> fooAsync(int x) {
  if {
      return CompletableFuture.completedFuture(0);
  } else {
      return CompletableFuture.supplyAsync(() -> doWork(x));
  }
}

int bar(int x) {
  return x;
}

int z = fooAsync(5).thenApply(x -> bar(x)).join();


/*
thenAccept(Consumer<? super T> action)

thenApply(Function<? super T, ? extends U> func) **Basically map**

thenCompose(Function<? super T, ? extends CompletableStage<U>> fn) **Basically flatMap**

theCombine(CompletionStage<? extends U> other,
           BiFunction<? super T, ? extends V> fn)

always end your code with a join()
```

```
jshell> CompletableFuture<Integer> cf = urls.stream().
   ...> map(x -> CompletableFuture.sypplyAsync(() -> processUrl(x))).
   ...> reduce(CompletableFuture.completedFuture(0), (x,y) -> x.thenCombine(y, (a,b) -> a + b))
```