CS2030 Programming Methodology II

AY2021/22 Semester 2

2 special ways to instantiate an SAM interface:

- Anonymous class:

```
Function<Integer, Integer> f =
    new Function<Integer, Integer>() {
        Integer apply(Integer x) {
            return x + 1;
        }
    };
```

- Lambda: `Function<Integer, Integer> f = x -> x + 1;`

SAM Interface

Producer Extends, Consumer Super (PECS)

`A<? extends T>` makes A a producer – clients can get things from A, but not put things into A;
`B<? super T>` makes B a consumer – clients can only put things into B, but not get things from B.

`Function<? super T, ? extends U>`

The input side of a function is a consumer, while the output side of a function is a producer.

`Function<T, U>`
`i/o: T -> U`
`x -> x.toString()`

`Function<T, Optional<U>>`
`i/o: T -> Optional<U>`
`x -> Optional.of(x.toString())`

`Supplier<? extends T>`
`i/o: () -> T`
`() -> 1`

`Consumer<? super T>`
`i/o: T -> ()`
`x -> { System.out.println(x); }`

`Runnable`
`i/o: () -> ()`
`() -> { System.out.println("empty"); }`

`UnaryOperator<T>`
`i/o: T -> T`
`x -> x + 1`

`Predicate<? super T>`
`i/o: T -> boolean`
`x -> x % 2 == 0`

`BinaryOperator<T>`
`i/o: (T, T) -> T`
`(x, y) -> x + y`

`BiFunction<T, U, R>`
`i/o: (T, U) -> R`
`(x, y) -> x.toString() + y.toString()`

Generic Class

```
class Box<T> {
    T item;

    Box(T t) {                          // Constructor
        this.t = t;
    }

    T get() {                           // Normal Method (return type)
        return this.item;
    }

    <U> Box<U> replace(U u) {           // Involving another type (new generic types to be used in this method)
        return new Box(u);
    }

    static <U> Box<U> of(U u) {         // Static Method
        return new Box(u);
    }
}
```

Declarative Programming

Optional

Source Operation

```
static <T> Optional<T> ofNullable(T t);
```
If `t` is not null, return `Optional[t]`; otherwise `Optional.empty`.

Intermediate Operation

```
<U> Optional<U> map(Function<? super T, ? extends U> mapper);
```
Use mapper to map `t` to `u`, then wrap it to be `Optional[u]`.

```
<U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper);
```
Use mapper to map `t` to `Optional[u]`, then return it directly.

Terminal Operation

```
T orElse(T t);
```
If optional is not empty, return the item inside; otherwise return `t`.

```
T orElseGet(Supplier<? extends T> supplier);
```
If optional is not empty, return the item inside; otherwise return `supplier.get()`.

```
T ifPresentOrElse(Consumer<? super T> action, Runnable
emptyAction);
```
If optional is not empty, run `action`; otherwise run `emptyAction`.

Stream

Source Operation

```
IntStream.range(m, n);
```
Create an `IntStream` from `m` to `n - 1`.

```
Stream.of(T t1, T t2, …);
```
Create an `Stream` containing the inputs.

```
Stream.iterate(T seed, UnaryOperator<T> f);
```
Create an `Stream` containing `seed`, `f(seed)`, `f(f(seed))`, etc.

```
list.stream();
```
Create an `Stream` from a `List`.

Intermediate Operation

```
<U> Stream<U> map(Function<? super T, ? extends U> mapper);
<U> Stream<U> flatMap(Function<? super T, Optional<U>> mapper);
Stream<T> filter(Predicate<? super T> predicate);
```
Filter the stream based on the given predicate.

```
Stream<T> limit(int size);
```
Create a new stream based on the first `size` element of a stream.

Terminal Operation

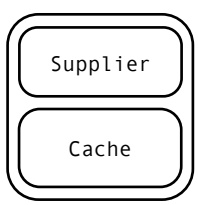
```
void forEach(Consumer<? super T> action);
```
Perform `action` on each element of the stream.

```
T reduce(T identity, BinaryOperator<T> accumulator);
```
Initially, `tempResult = identity;`
For each `element` in the stream:
`tempResult = accumulator.apply(tempResult, element);`
e.g. `Stream.of(1, 2, 3).reduce(0, (x, y) -> x + y);`

```
<U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator,
BinaryOperator<U> combiner);
```
Initially, `tempResult = identity;`
For each `element` in the stream:
`tempResult = accumulator.apply(tempResult, element);`
OR
`tempResult = combiner.apply(tempResult1, tempResult2);`

```
e.g. Stream.of("1", "2", "3").reduce(0,
         (a, s) -> a + Integer.parseInt(a) + s),
         (x, y) -> x + y);
```

Lazy



Only when the first time when `.get()` method is called, the `supplier` is run and we store the result in to cache.

Cache is initially `Optional.empty`. When the `supplier` is run, we store it to the `Optional`.

```
PriorityQueue pq = [1, 2]
```

```
o1 = Optional.of(pq.poll())
o2 = Optional.of(pq.poll())
o1.get()
o2.get()
```

VS

```
l1 = Lazy.of(() -> pq.poll())
l2 = Lazy.of(() -> pq.poll())
l1.get()
l2.get()
```