CS2030S PE 2

Assign	
Progress	
Series	
Tags	
Type	Notes

- **@SafeVarargs** — only items of a specific type going into an array (used to circumvent the need for @SuppressWarnings(“unchecked”)

  ```
  @SafeVarargs
  public static <T> Stream<T> of(T... args) {
    // args is a T[]
    List<T> l = new ArrayList<>();
    for (T el : args) l.add(el);
    return new Stream<>(l);
  }
  ```

- **T… items** — variable number of arguments of the same type (see above)
- **@FunctionalInterface** — used to indicate that an interface is going to be used as a functional interface; throws an exception if more than one abstract method is found in the interface

  ```
  @FunctionalInterface
  interface Predicate<T> {
    boolean test(T input);
  }

  public Stream<T> filter(Predicate<T> pred) { }

  stream.filter(input -> input > 10)
  ```

- Common functional interfaces **(java.util.function)**
  - **Predicate<T>::test** — boolean test(T t) (see above)
  - **Consumer<T>::accept** — void accept(T t)

    ```
    Consumer<Integer> l = (v) -> System.out.println(v);
    l.accept(15);  // prints ja15
    ```

  - **Supplier<T>::get** — T get()

```
    Supplier<Integer> i = () -> 15;
    foo(i.get());  // same as foo(15)

    Supplier<Integer> l = () -> {
      System.out.println("supplier");
      return 15;
    }
    foo(l.get());  // prints "supplier" too
    ```

  - **Function<T, R>::apply** — R apply(T t)

    ```
    Function<Integer, String> f = i -> "i is " + i;
    f.apply(15);  // returns "i is 15"
    ```

  - **UnaryOperator<T>::apply** — T apply(T t) (represents an operation on a single operand that produces a result of the same type as its operand)

    ```
    UnaryOperator<Integer> u = i -> i + 1;
    u.apply(15);  // returns 16
    ```

  - **BinaryOperator<T>::apply** — T apply(T t1, T t2) (inherits from BiFunction below)

    ```
    BinaryOperator<Integer> bi = (x, y) -> x + y
    bi.apply(15, 23);  // returns 38
    ```

  - **BiFunction<S, T, R>::apply** — R apply(S s, T t)

    ```
    BiFunction<Integer, String, String> b = (i, s) -> i + s;
    b.apply(1, " is the best");  // returns 1 is the best
    ```

- Common abstractions
  - java.util.Optional<T>
    - Optional.of(T t)
    - T optional.get()
  - java.util.stream.Stream<T>
- Streams API
  - Note: bounded — only works on finite streams
  - Note: streams can only be operated on once, recreate if need to operate on it again
  - **java.util.stream**
  - Creating a stream

    - Static factory method — **Stream.of(**…**args)**
    - **Stream::generate** — Stream<T> generate(Supplier<T> s)

```
Stream.generate(() -> 5);
```

    - **Stream::iterate** — Stream<T> iterate(T seed, UnaryOperator<T> f)

```
Stream.iterate(1, i -> i * 2);
```

    - **Arrays::stream** — Stream<T> stream(T[] array)

```
Stream<T> foo(T... args) {
  return Arrays.stream(args);
}
```

      - **Arrays::asList** — List<T> asList(T… a) (can pass varargs as is)

```
List<T> of(T... args) {
  return Arrays.asList(args);
}
```

    - **List::stream**
    - **List::parallelStream**
  - Common operations
    - **flatMap** — transforms every element in the stream into another stream with the resulting stream of streams being flattened and concatenated together
      - Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)

```
// returns stream of (2, 3, 4, 4, 6, 8, 6, 9, 12)
List.of(1, 2, 3).stream().flatMap(t -> List.of(t * 2, t * 3, t * 4).stream())
```

    - **map** — transforms every element in the stream into another stream (nested streams are not flattened this way)
      - Stream<R> map(Function<? super T, ? extends R> mapper)

```
// returns stream of (2, 4, 6)
List.of(1, 2, 3).stream().map(i -> i * 2)
```

- **sorted** — returns stream with the elements in the stream sorted [bounded]
  - Stream<T> sorted() — defaults to ascending order
- **distinct** — returns a stream with only distinct elements in the stream [bounded]
  - Stream<T> distinct()
- **limit** — returns a stream containing the first n elements of the stream
  - Stream<T> limit(long maxSize)

  ```
  // returns stream of (1, 2, 3)
  List.of(1, 2, 3, 4, 5).stream().limit(3)
  ```

- **takeWhile** — returns a stream containing the elements of the stream until the predicate becomes false (can remain infinite)
  - Stream<T> takeWhile(Predicate<? super T> predicate)

  ```
  // returns stream of (1, 2, 3)
  List.of(1, 2, 3, 4, 5).stream().takeWhile(x -> x < 4)
  ```

- **peek** — apply a lambda on a “fork” of the stream (allow side effects without affecting the stream)
  - Stream<T> peek(Consumer<? super T> action)

  ```
  // prints 1..5 while mapping to stream of (2, 4, 6, 8, 10)
  List.of(1, 2, 3, 4, 5).stream().peek(System.out::println).map(x -> x * 2)
  ```

- **reduce** — apply a lambda repeatedly on the elements of the stream to reduce it into a single value (first argument is the accumulator, the second is the current value in the stream)
  - T reduce(T identity, BinaryOperator<T> accumulator)

    ```
    // returns 1 * 1 * 2 * 3 * 4 * 5
    List.of(1, 2, 3, 4, 5).stream().reduce(1, (acc, cur) -> acc * cur)
    ```

  - <U> U reduce(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner) — used in parallel streams to combine different sub-streams into one (the **accumulator** acts as a map and then each element is combined into one whole element with **combiner**)

    ```
    // stream is Stream<Character>
    // this reverses the character stream
    ```

```
stream
  .map(ch -> ch.toString())
  .parallel()
  .reduce(
      "",
      (acc, cur) -> cur + acc,       // run on each sub task
      (sub1, sub2) -> sub2 + sub1);  // combines the result of each sub task
```

        - If used sequentially, the equivalent of performing a map first then reduce

```
// returns "\nnums: 1\nnums: 2\nnums: 3"
List.of(1, 2, 3)
  .reduce(
      "",
      (acc, cur) -> acc + "\nnum: " + cur,
      (sub1, sub2) -> sub1 + sub2);

// equivalent to
List.of(1, 2, 3).map(x -> "\nnum: " + x).reduce("", (acc, cur) -> acc + cur);
```

    - **filter** — returns a stream with only the elements that pass the filter
        - Stream<T> filter(Predicate<? super T> predicate)

```
// returns stream of (2, 4)
List.of(1, 2, 3, 4, 5).filter(x -> x % 2 == 0)
```

    - **noneMatch** — returns true if none of the elements pass the given predicate
    - **allMatch** — returns true if every element passes the given predicate
    - **anyMatch** — returns true if at least one element passes the given predicate
    - **parallel** — parallelize the stream (order matters)
    - **sequential** — marks the stream to be sequential (order matters)
    - **collect**(Collectors.toList()) — converts stream into a List<T>
    - **unordered** — convert an ordered stream into an unordered stream (to avoid parallel() from optimizing for order)
  - Common types of streams
    - IntStream
        - IntStream::range(x, y) — generates integer stream from x to y - 1
    - LongStream
    - DoubleStream
- Monad
  - Key properties:

        - of — initialising the value and side information
        - flatMap — update the value and side information
    - Laws to follow
        - Identity laws
            - Left: Monad.of(x).flatMap(x -> f(x)) — must be the same as f(x)
            - Right: monad.flatMap(x -> Monad.of(x)) — must be the same as monad
        - Associative laws — monad.flatMap(x -> f(x)).flatMap(x -> g(x)) == monad.flatMap(x -> f(x)).flatMap(y -> g(y))
- Functor: ensure that lambdas can be applied sequentially to a value without worrying about side effects
    - Preserving identity — functor.map(x -> x) == functor
    - Preserving composition — functor.map(x -> f(x)).map(x -> g(x)) == functor.map(x -> g(f(x)))
- Threads
    - Thread-safe data structures
        - java.util.concurrent.CopyOnWriteArrayList
    - Creating threads
        - java.lang.Thread
        - new Thread(() -> {}).start()
    - Common operations
        - Thread.currentThread().getName()
        - Thread.sleep(n)
        - Thread::isAlive
    - java.util.concurrent.CompletableFuture — monad to perform tasks concurrently
        - CompletableFuture::thenComposeAsync(x -> {})
        - CompletableFuture::thenComposeAsync(CompletableFuture, (x, y) -> {})
        - Creating CompletableFuture
            - CompletableFuture.completedFuture(x) — task is already completed and return the value x
            - CompletableFuture.runAsync(() -> {}) — task to complete when lambda (Runnable) finishes, returns CompletableFuture<Void>

      - CompletableFuture.supplyAsync(() -> (T) x) — task completes when lambda expression finishes, returns CompletableFuture<T>
      - CompletableFuture.allOf(… CompletableFuture<T>) — only completed when every given CompletableFuture completes
      - CompletableFuture.anyOf(… CompletableFuture<T>) — completed when any one of the given CompletableFuture completes
    - Common operations
      - thenApply[Async] — map
      - thenCompose[Async] — flatMap
      - thenCombine[Async](CompletableFuture, (x, y) -> {}) — combine
      - get — get the result after all CompletableFutures are completed (with exceptions)
      - join — similar to get without any exceptions
      - handle((value, exception) -> return value) — exception handling (continue chaining tasks even with exceptions)
- Thread pools
  - java.util.concurrent.ForkJoinPool
  - java.util.concurrent.RecursiveTask<T> — task that can be forked and joined (to be used by ForkJoinPool)
    - Override T compute() to compute the value of the sub-task
    - Fork one side, compute the other (which further forks the problem) — we can leave this task to be performed in the main thread, join the side that was forked (the other task is performed in a different thread)
    - Call join() on the task that was the last to be forked