1 Encapsulation

Definition 1.1 *The practise of containing different properties of a certain object as class variables.*

2 Information Hiding

Definition 2.1 *The practise of hiding variables behind an abstraction barrier using the `private` keyword.*

3 Tell, Don't Ask

Definition 3.1 *The principle where accessors and mutators such as `get()` and `set()` are not used.*

4 Polymorphism

Definition 4.1 *Dynamic binding occurs when the compile-time type of an object is different from its run-time type.*

For example, overriding the `equals` method of a class allows us to call the new method even when the object's compile-time type is `Object`.

```
boolean contains(Object array[], Object obj) {
    for (Object curr : array) {
        if (curr.equals(obj)) {
            return true;
        }
    }
    return false;
}
```

5 Liskov's Substitution Principle

Definition 5.1 *Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(x)$ should be true for objects y of type S where $S <: T$.*

Hence, we may want to mark classes as `final` to prevent inheritance that breaks LSP.

6 Abstract Classes

Definition 6.1 *A class with abstract methods that cannot be instantiated directly, and has to be extended from.*

Abstract classes uses the keyword `abstract` for its abstract methods.

7 Interfaces

Definition 7.1 *An interface is a type and models what an entity can do.*

If class C implements an interface I, $C <: I$ where C is a sub-type of I.

7.1 Impure Interfaces

Definition 7.2 *An interface that has concrete implementations using the `default` keyword. Classes that implemenet this interface will all have this default method unless it is overidden.*

8 Wrapper

8.1 Auto-boxing and unboxing

Definition 8.1 *Automatic conversion between wrapper types and primitive types, such as `Integer` to `int`.*

8.2 Performance

Poorer performance as more memory is required to instantiate and collect garbage.

9 Immutability

Definition 9.1 *A variable or class is immutable if it has the `final` keyword. It disallows mutation or inheritance.*

Advantages for immutability:

- **Ease of understanding.**

  Ensures that there is no mutation along the way when an object is passed around many times.

- **Enabling safe sharing of objects.**

  Objects can be reused without instantiating new objects.

- **Enables safe sharing of internals**

  `Varargs` or `T...` allows for the passing of variable arguments to as a parameter. It is as good as passing in an array.

- **Enable safe concurrent execution.**

  Important when threads are running in parallel to avoid side effects and unintended mutations or an incorrect sequence of mutation to occur.

10 Nested Classes

Definition 10.1 *A class defined within another containing class.*

- Nested classes **can** access private fields of the container class.
- Can either be `static` or non-`static`.
- Static nested classes are associated with the **containing class** and can only access static variables and methods.
- **Qualified State:** A `this` reference with a prefix of the enclosing class. (e.g `A.this.x`).

10.1 Local Class

Definition 10.2 *Class defined locally within a function.*

```
void sortNames(List<String> names) {
    class NameComparator implements
        Comparator<String> {
        public int compare(String s1, String s2) {
            return s1.length() - s2.length();
        }
    }
    names.sort(new NameComparator());
}
```

10.2 Variable Capture

Definition 10.3 *Local classes makes a copy of local variables and captures the local variable. Local classes can only access `final` variables.*

10.3 Anonymous Class

Definition 10.4 *A class that is declared and instantiated in a single statement.*

```
names.sort(new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.length() - s2.length();
    }
});
```

11 Functions

We can refer to functions in classes with the `::` operator (e.g `List::sort`).

11.1 Pure Functions

- A pure function does not cause any side effects. It must **only** return a value given an input, and do nothing else.
- A pure function must be deterministic. A function must **always** produce the same output given the same input.

11.2 Lambda Functions

If we use the {} brackets after the `->` arrow, we need to specify which line to `return`.
There is no need to specify the type when we specify a function interface, such as `Function<S, T>` or `BiFunction<S,T,R>`.

12 Streams

- `Predicate<T>::test`
- `Supplier<T>::get`
- `Function<T,R>::apply`
- `UnaryOp<T>::apply`
- `BiFunction<S,T,R>::apply`

Stream Operations:

`forEach`	Applies a lambda to each element. Terminal.
`flatMap`	Applies a function and transforms into another stream but flattens it.
`limit`	Returns a trunctated stream with the first n elements.
`takeWhile`	Returns a truncated stream up till the first fails the input predicate.
`peek`	Takes in a consumer and returns a new stream with that operation.

13 Monad

Laws:

- Identity Law
- Associative Law

13.1 Identity Law

Left Identity Law: `Monad.of(x).flatMap(x -> f(x))` = `f(x)`.
Right Identity Law: `monad.flatMap(x -> Monad.of(x))` = `monad`.

13.2 Associative Law

`monad.flatMap(x -> f(x)).flatMap(x -> g(x)))` = `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`.

13.3 Functors

Definition 13.1 *A functor is a simpler construction than a monad in that it only ensures lambdas can be applied sequentially to the value, without worrying about side information.*

Laws:

- `functor.map(x -> x) == functor`.
- `functor.map(x -> f(x)).map(x -> g(x)) == functor.map(x -> g(f(x))`.

14 Parallel Streams

All parallel programs are concurrent, but not all concurrent programs are parallel.
`parallel()` can be called on streams to parallelise it. The opposite call is `sequential()`. The latest call overrides all the previous calls.

14.1 Parallelis-ability

14.1.1 Interference

Definition 14.1 *Interference means that one of the stream operations modifies the source of the stream during the execution of the terminal operation.*

```java
List<String> list = new
    ArrayList<>(List.of("Luke", "Leia", "Han"));
list.stream()
    .peek(name -> {
        if (name.equals("Han")) {
        list.add("Chewie"); // they belong
            together
        }
    })
    .forEach(i -> {});
```

14.1.2 Stateful vs Stateless

Definition 14.2 *A stateful lambda is one where the result depends on any state that might change during the execution of the stream.*

```java
Stream.generate(scanner::nextInt)
    .map(i -> i + scanner.nextInt())
    .forEach(System.out::println)
```

14.1.3 Side Effects

In the code below, `forEach` modifies the arrayList and an incorrect sequence may arise.

```java
List<Integer> list = new ArrayList<>(
    Arrays.asList(1,3,5,7,9,11,13,15,17,19) );
List<Integer> result = new ArrayList<>();
list.parallelStream()
    .filter(x -> isPrime(x))
    .forEach(x -> result.add(x));
```

Solution: Use thread-safe methods or data structures like `.collect` or `CopyOnWriteArrayList<>`.

14.1.4 Associativity

`reduce` is parallelisable, but the function must be associative. Consider:

```java
Stream.of(1,2,3,4).reduce(1, (x, y) -> 1 * x + y)
```

Rules:

- `combiner.apply(identity, i) == i`
- `combiner` and `accumulator` must be associative.
- `combiner` and `accumulator` must be compatible – `combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)`.

14.2 Order

- `findFirst`, `limit` and `skip` is expensive on an ordered stream as coordination is required to maintain order.
- `unordered()` can be called on streams where order is not important to speed up the parallelisation.

15 Threads

Java has a class `java.lang.Thread` that can be used to encapsulate a function to run in a seperate thread.

```java
new Thread(() -> {
  for (int i = 1; i < 100; i += 1) {
    System.out.print("_");
  }
}).start();

new Thread(() -> {
  for (int i = 2; i < 100; i += 1) {
    System.out.print("*");
  }
}).start();
```

The `.parallel()` call splits a stream operation into multiple threads.
`Thread.sleep()` can be called on a thread to add a delay to a thread before it is ready to run again.

16 Async

`Thread` is a low-level abstraction that is not very easy to use, a higher level abstraction would be `CompletableFuture<T>`.

`completedFuture`	Creates a task that is completed, returns T.
`runAsync`	Takes in `Runnable` and returns `CompletableFuture<Void>`.
`supplyAsync`	Takes in `Supplier<T>` and returns `CompletableFuture<T>`.
`thenApply`	Map function that runs when the `CompletableFuture` instance is completed.
`thenCompose`	flatMap function that runs when the `CompletableFuture` instance is completed.
`thenCombine`	combine function that runs when the `CompletableFuture` instance is completed.
`get`	Blocks all operations until the given `CompletableFuture` is completed and returns the value.
`join`	Same as `get` but no checked exception is thrown.
`exceptionally`	Takes in a function that takes a `Throwable` and returns a new type S. Returns `CompletableFuture<S>`.

17 Fork and Join

Java has an implementation `ForkJoinPool` that is fine-tuned for the fork-join model.

It is a parallel divide-and-conquer mode of computation. There is a `compute` method to implement. The class `RecursiveTask<T>` supports the methods `fork`, `join` and `compute`.

```java
class Summer extends RecursiveTask<Integer> {
    private static final int FORK_THRESHOLD
        = 2;
    private int low;
    private int high;
    private int[] array;

    public Summer(int low, int high, int[] array)
        {
        this.low = low;
        this.high = high;
        this.array = array;
    }

    @Override
    protected Integer compute() {
        // stop splitting into subtask if array
            is already small.
        if (high - low < FORK_THRESHOLD) {
            int sum = 0;
            for (int i = low; i < high; i++) {
            sum += array[i];
            }
            return sum;
        }

        int middle = (low + high) / 2;
        Summer left = new Summer(low, middle,
            array);
        Summer right = new Summer(middle,
            high, array);
        left.fork();
        return right.compute() + left.join();
    }
}
```

- Each thread has a queue of tasks.
- If a thread is idle, it checks the queue and picks a task at the had and executes it. If the queue is empty, it finds another queue that is non-empty to dequeue from.
- When `fork` is called, the caller adds itself to the head of the queue
- When `join` is called:

  If the subtask to join has not been executed, then `compute` is called on the subtask.

  If the subtask is completed, then the result is read and `join` returns.

17.1 Order of fork() and join()

We should `join()` the most recently `fork()`-ed task first since `ForkJoinPool` adds and removes tasks from the queue in the order in which we call `fork` and `join`.
Fast Example:

```java
left.fork();
right.fork();
return right.join() + left.join();
```

Slow Example:

```java
left.fork();
right.fork();
return left.join() + right.join();
```