Async Programming

- Cf async runs as a daemon thread (if no executor is provided) on the ForkJoinPool. JVM can exit main thread if the only other threads left are daemon threads (non-user created threads)
- As a result, tasks running on daemon threads may not complete fully (e.g. printing results stops halfway). This may happen if you do not call `join()` to wait for a `cf` to finish
- Calling `join()` after `anyOf()` does the same thing – the task that finishes second is not guaranteed to finish completing before the main method returns
- `join()` is blocking and forces main thread to wait for results of other thread, so the thread being joined is guaranteed to execute fully
- Forked tasks are added to the head of the current thread's deque. Forkable tasks are subclasses of the abstract class `RecursiveTask<T>` and implement the `compute()` method. Inside the compute method, tasks are broken down using `fork()` and `join()`. An idle thread can **steal work from the back of another thread's deque** if it's own deque is empty. You should call `join()` on **latest tasks** that have been forked since they are at the head and will start executing first
- Calling `compute()` runs the computation on the *current* thread.
- Chaining multiple tasks on the *same instance* of `cf` makes them run in **parallel** (no guarantee of which executes first). Chaining them one after another guarantees that one is completed before the other
- `RunAsync` and `SupplyAsync` are considered complete when the **given lambda expression completes** (after the call to `run()` for Runnable or `get()` for Supplier finishes). `RunAsync` takes in a `Runnable`, `SupplyAsync` takes in a `Supplier`
- `ThenAccept` takes in a `Consumer` and returns `CF<Void>`
- `completedFuture()` is considered complete when the function call to `completedFuture()` returns
- `join()` does not throw checked exceptions, while `get()` throws checked exceptions. Both are blocking
- `ThenCombine` waits for the other `CF` to complete before combining the two results. Produces `CF<V>`

Monads

- Left Identity Law: `Monad.of(x).flatMap(x -> f(x))` must be the same as `f(x)`
- Right Identity Law: `monad.flatMap(x -> Monad.of(x))` must be the same as `monad`
- Associativity Law (no matter how you group the flatmap, the result must be the same): `monad.flatMap(x -> f(x)).flatMap(x -> g(x))` must be the same as `monad.flatMap(x -> f(x).flatMap(y -> g(y)))`
- `Functor.of(x).map(x->f(x))=Functor.of(f(x))` is not guaranteed to hold true, e.g. `Maybe.of(null)` is `None` but `Maybe.some(null)` is `Some`

Overriding

- **Return type** must be **covariant** (same type or subtype) as return type in parent method
- Access modifier must be **equal** or **more accessible** than parent
- Cannot throw a checked exception that broader than the one in the parent method
- Cannot throw a checked exception if parent method does not throw a checked exception
- It is OK for child method to **not** declare any checked exception even if parent method throws a checked exception
- Unchecked methods (subclasses of `RuntimeException`) can still be declared
- If you mark a method as `@Override` but override it incorrectly, the compiler will throw a compile error
- **Private methods are not inherited**. You can declare the same method name and signature in a child class
- Static methods are not inherited, though they can be *hidden* by a child class. The method in the child class has to be marked `static` and have the same method signature as that in the parent class
- **Final** methods **cannot be overridden** by child classes
- `A<R>` and `A<? extends R>` both erase to A after type erasure, but they are considered different method parameters. So if the parent method uses `A<R>`, the child method must also override with `A<R>`. If the parent method uses `A<? extends R>`, the child method must also override with `A<? extends R>`

Overloading

- Be careful of clashing method signatures when using generic method parameters. **After type erasure, only the class type remains**. If the method signature (name + parameter types + order and number of params) after type erasure clashes, the compiler will throw a compile error. Return type is **not** a part of *method signature*, but it is part of the *method descriptor*.
- `A<R>` and `A<? extends R>` still erase to the same parameter, so they cannot be overloaded together

Liskov Substitution Principle

- When answering questions about LSP, answer in the form "Yes, LSP is violated..." or "No, LSP is not violated" + (reason), in the format "The desirable property of ___ is/is not changed"

Stack and Heap

- Static fields are drawn in the metaspace
- New objects are created in the heap
- Method calls and local variables are drawn in the stack space
- Instance methods on the stack have a variable *this* pointing to the object the method is called on
- Strings are stored in the *intern pool* (which is also in the heap), and they are reused unless specifically a *new* String is being created.
- Arrays are drawn as 0-indexed boxes with arrows pointing to objects (or containing literal values)
- Lambda functions are also added to the stack when they are called. Lambda functions themselves are **heap-allocated**.
- Variables declared 'earlier' will appear lower in the stack, while variables declared 'later' will appear higher in the stack

Exceptions

- Exceptions must be caught in order of how broad they are – e.g. you cannot catch `Exception` before `IOException` (compile error). The **broadest** exception type should be caught **last**. But if there is no subtyping relation between the exceptions being caught, the order does not really matter (e.g. two separate subtypes of `Exception`)
- Bad design: exceptions should not be used for flow control. Only if/else/while should be used
- Bad design: Pokemon "catch all" exception that catches all possible exceptions is not very useful or informative
- **Checked exceptions** are outside of programmer's control, e.g. `FileNotFoundException`. **Unchecked exceptions** are due to *logical error*, e.g. `NullPointerException`
- Be careful of tricky print statements that come after an exception is thrown

Capturing Variables

- Only variables **on the stack** *need* to be captured since they will be gone after the method returns; variables **on the heap** *do not need* to be captured since they persist until they are garbage-collected

Nested Classes

- 4 types: inner class, static nested class, local class, anonymous class

- Inner class: `outer.new Inner()`. If inner class is `private`, any code outside of the class cannot access the private inner class, instantiate it, refernce its type, or use the methods of the inner class. However, the outer class can still return *references* to the inner class
- Static class: `new Outer.Inner()`.
- Local class only captures local variables needed for its class definition. Variables defined within the local class are not captured.
- An anonymous class can only extend **ONE** class/abstract class or interface. It cannot be an anonymous class of multiple classes and interfaces at the same time. Anonymous classes can extend generic classes e.g. `Comparator<Integer>`

Reminders

- Abstract classes and interfaces **cannot be directly instantiated** – they need a concrete class to implement it
- When a concrete class implements an interface or extends an abstract class, they need to override ALL the abstract methods, otherwise compiler will throw an error
- Remember that `this` keyword **cannot** be used in `static` methods, only instance methods
- Java can only do either autoboxing or casting, but not both together
- Functional interfaces have a **S.A.M** (single abstract method)

Immutability

- Immutable objects are **easier to reason with**, enable **safe concurrent execution**
- Final is **insufficient** to make a class immutable, but it is **necessary**.
- Immutability allows **safe sharing of internals**: creating a copy of the object each time it is requested
- `@SafeVarargs` is used to suppress compiler warning about varargs, as Java cannot have generic arrays. Varargs are just syntactic sugar for arrays, so arrays can be passed into vararg parameters. Be careful of `null` with varargs. `(null, null)` is inferred as array of length 2. `(null)` inferred as null, not array. `((Object) null)` is inferred as array of 1 object.
- If the caller still has a reference to an object passed into a class's constructor, the class may not be final since the caller can still modify the object from outside the class.
- `Final` classes cannot be extended, `final` methods cannot be overriden
- All primitive wrapper classes are immutable. Using `Integer` in a for loop is inefficient as it causes a new instance of `Integer` to be created for each iteration. Primitive types can be autoboxed to fit methods with `Object` parameters.

OOP Principles

- Tell, Don't Ask: A client should tell an object what task to perform, rather than asking for the data from the object and performing the task itself.
- Inheritance: Subclassing/Subtyping, avoid repeated code, allow overriding
- Composition/Encapsulation**:** Class has reference to object of another class. Bundling of related data together in the same class
- Information Hiding/Abstraction: Use access modifiers like private to hide fields that are not relevant to the user
- Polymorphism: Dynamic binding behaviour at runtime

Method Reference

- Static methods: `Math::abs` inferred as `x -> Math.abs(x)`
- Instance methods of a particular object: `str::charAt` inferred as `i -> str.charAt(i)`
- Instance methods of an arbitrary object of a particular type: `String::concat` inferred as `(s1, s2) -> s1.concat(s2)`
- Constructors: `HashSet<Integer>::new` inferred as `() -> new HashSet<Integer>()`

Typing

- Strongly Typed: each variable needs to have a declared type
- `x instanceof y` checks if the RTT of `x` is a subtype of `y`
- For dynamic binding, the RTT of the target, the compile-time type of method parameters, and the return type of the method invoked is used.
- Covariant: if S<:T, C(S) <: C(T). Contravariant: if S<:T, C(T)<:C(S). Java arrays are covariant.
- Casting is a **run-time operation.**

Generics

- The **type parameter** is specified in the brackets <>
- When a type parameter is **instantiated**, it is called a **parameterized type**. E.g. `A<String>`
- **Bounded type parameters** (`? extends .../? super ...`) allows us to put constraints on type parameters.
- **Type erasure** is performed during **compile-time. ALL** traces of <> and generic types are removed.
- After type erasure, the compiler **inserts** a typecast to cast the return type into the desired type.
- `@SuppressWarnings` should be used responsibly at the *most limited* scope, and a comment should explain why it is safe to suppress the warnings. It can only be used on an **declaration** statement.
- You should always avoid using raw types and use `<?>` instead

Currying Functions

```
public static <S,T,R> Transformer<S, Transformer<T, R>>
curry(Combiner<S, T, R> f) { return x -> y -> f.combine(x,
y); }

public static <S,T,R> Combiner<S,T,R>
uncurry(Transformer<S, Transformer<T, R>> f) { return (x,
y) -> f.transform(x).transform(y); }
```

InfiniteList<T>

(Using Producer instead of Lazy)

```
public Producer<T> lazyGet(int i) { return () -> i == 0 ?
this.head() : this.tail().lazyGet(i - 1).produce(); }

private InfiniteList<T> skip(T x) { return
this.head().equals(x) ? this.tail().skip(x) : this; }

public InfiniteList<T> unique() { return new
InfiniteList<>(() -> this.head(),() ->
this.skip(this.head()).unique());}
```

(Using Lazy instead of Producer)

```
public InfiniteList<T> append(InfiniteList<T> list)
{ return new InfiniteList<T>(head, tail.map(t ->
t.append(list))); }

public <U, R> InfiniteList<R> zipWith(InfiniteList<U> list,
Transformer<T, Transformer<U, R>> transform) { return new
InfiniteList<R>( head.map(transform).map(h ->
h.transform(list.head())), tail.map(t ->
t.zipWith(list.tail(), transform))
); }
Take note when the given transformer is a curried function
instead of BiFunction or Combiner

public <R> InfiniteList<R>
flatMap(Transformer<T, InfiniteList<R>> transformer) {
Lazy<InfiniteList<R>> list = this.head.map(transformer);
return new InfiniteList<>(
      list.map(l -> l.head()),
      list.map(l ->
l.tail().append(tail().flatMap(transformer)))
  );
}
```