**Java**

- Statically-typed (variable types are known at compile time)

- strong-typed language (rules enforcing type safety)

**Principles of OOP**

Abstraction

- Hides away unnecessary complexity

- Helps us to think at a higher conceptual level without worrying about the details

**-**\_HAS-A relationship : use composition

• IS-A relationship : use inheritance

**Polymorphism**

- Allows us to change how existing code behaves, without changing a single line of the existing code (or even having access to the code)

- Changes based on run-time type

o See Dynamic-Binding for more information

**Final vs Static**

Instance Fields: Associated with an object  
 Class Fields: static keyword

Reference Types vs Primitives

Primitive types (holds numeric/boolean values)

- byte <: short <: int <: long <: float <: double

- char <: int

Reference types (stores only the reference to the value)

**Tell, Don’t Ask**

- Implement a method within the class that does whatever we want the class to do

- The client should tell object what to do, instead of asking for

information/value of a field then perform the computation on the

object's behalf.

- Keeps encapsulation intact Composition (HAS-A Relationship)

- Applies principles of abstraction and encapsulation

**Aliasing:**

- Reference types sharing same reference values

- Another approach to address this is immutability

**Method Signature vs Descriptor**

Method signature: Method name and the number, type, and order of its parameters [RETURN TYPE **C::foo(B1, B2)]**

Method descriptor: Method signature plus the return type.

**Overriding**

o Static methods cannot be overridden

**Overloading**

- Two or more methods in the same class with the same name but a differing method signature

- Allows us to write methods to handle differing inputs

- As it is also a method, it is possible to overload the class constructor as well

**Dynamic Binding (or late binding or dynamic dispatch)**

- Java allows type casting from type to if S< T .This is called narrowing type conversion.

- Only applies to instance methods

- The method to invoke for class methods is resolved statically

During compile time

- If one method's parameter type is a subtype of another method's parameter type, the more specific method is the one with the subtype parameter type.

**Compile Time:**

- Using object (this) compile-time type, the compiler searches for all

methods that can be correctly invoked on the given argument to

determine the method descriptor

- Most specific matching method that accepts argument

**Run Time:**

- Method descriptor (previously determined) is retrieved

- Java looks for an accessible method with the matching descriptor in run-time type. If no such method is found, the search will continue up the class hierarchy

- The first method implementation with a matching method descriptor found will be the one executed.

**Liskov Substitution Principle**

- Part of inheritance/polymorphism principle

- Ensures any inheritance with method overriding does not introduce bugs to existing code

- LSP cannot be enforced by the compiler. The properties of an object have to be managed and agreed upon among programmers

- A subclass should not break the expectations set by the superclass. If a class B is substitutable for a parent class A then it should be able to pass all test cases of the parent class A.

- “final” keyword prevents classes from being inherited and methods from being overridden

**Abstract Class**

- A class that has been made into something so general that it cannot and should not be instantiated

o An abstract method cannot be implemented and therefore should

not have any method body.

**Interface (Models CAN-DO behaviour, -able suffix)**

- All methods declared in an interface are public abstract by default

- For a class to implement an interface and be concrete, it has to override all abstract methods from the interface and provide an implementation to each

o otherwise, it becomes abstract

- Java allows casting with interfaces as there is the possibility that a subclass could implement the interface and therefore Java allows it to compile

- Impure interfaces have some abstract methods and some non-abstract default methods

**Wrapper Classes (Reference Types)**

- class that encapsulates a type, rather than fields and methods.

o Immutable

- using an object comes with the cost of allocating memory for the object and collecting ‘garbage’ afterward, making it less efficient than primitive types

**Auto-boxing / Auto-unboxing**

- perform type conversion between primitive type and its wrapper class

**Casting**

- Making an explicit cast to a subtype is a narrowing type conversion

- Essentially asking the compiler to trust that the object returned has a run-time type of whatever it’s cast to (or its subtype)

**Variance**

- covariant if S <: T implies C(S) <: C(T)

- contravariant if S <: T implies C(T) <: C(S)

- invariant if it is neither covariant nor contravariant.

- Arrays of reference types are covariant

**Exceptions**

“throws” keyword declares that the construct is throwing an exception.

- causes the method to immediately return

**Try/Catch/Finally**

- Try block contains the normal code to run

- Catch block handles exceptions (instances that are subtypes of

Exception class)

o Information about an exception is encapsulated in an exception instance

- Finally block executes ‘housekeeping tasks’

-Exceptions are reference types.

o always executed even when return or throw is called in a catch block.

**Checked vs Unchecked Exceptions**

- An unchecked exception is an exception caused by a programmer's errors.

o Are not explicitly caught or thrown.do not require a try-catch block.

- A checked exception is an exception that a programmer has no control over

o should thus actively anticipate the exception and handle them, or

else the code would not compile, therefore anticipates it/

- A good program always handles checked exception gracefully and hides the details from the users.

**Good Practices:**

- Catch exceptions to clean up (resource deallocation)

- Avoid Pokemon Exception Handling (catching superclass)

- Avoid overreacting (exiting program)

- Do not break the abstraction barrier

o handle the implementation-specific exceptions within the

abstraction barrier

- Do NOT use Exception as a Control Flow Mechanism

\*Error class is used for situations where the program should terminate as

generally there is no way to recover from the error

**Generics (e.g. Object<Type>)**

- A generic type that takes other types as type parameters

o e.g. class Pair<S, T> {…} into Pair<String, Integer>

- Passing in type arguments instantiates into parameterised type

o Only reference types can be used as type arguments

- Generics are INVARIANT Methods

- Type parameter declared before return type of method

o e.g. <T> boolean contains(T[] array) {…}

PECS (Producer extends, Consumer super)

- ‘extends’ keyword to indicate T being a subtype of S

o acts as a producer of variable of type T o e.g. <T extends S>

- ‘super’ keyword to indicate T being a supertype of S

o acts as consumer of variable of type T e.g. <T super S>

**Type Erasure**

- Generic types are bounded during compilation

o <T> replaced with <Object>, or (if bounded), replaced by the

bounds instead

- Heap pollution: situation where a variable of a parameterized type refers to an object that is not of that parameterized type. Type-checking is only done during run-time

\* An array is what is called reifiable type -- a type where full type

information is available during run-time.

Generics are NOT reifiable(can be concerete) due to type erasure

**Unchecked Warnings**

- due to type erasure, there could be run-time errors that the compiler cannot prevent - also don’t use raw types thanks (see Wildcards)

**Wildcards <?>**

- A substitute for any type, Seq<?> is a supertype of every parameterized type of Seq<T>

- Can be upper-bounded (<? extends T>) or lower-bounded (<? super T>) Type Inference

-In resolving the type constraints for a given type parameter T:

-Seq<?> is a sequence of objects of some specific, but unknown type;

-Seq<Object> is a sequence of Object instances, with type checking by the compiler;

-Seq is a sequence of Object instances, without type checking.

-Whilst Object is the supertype of all T, it does not follow that Seq<Object> is the supertype of all Seq<T> due to generics being invariant.

**Type Inference**

- Type1 <: T <: Type2, then T is inferred as Type1

- Type1 <: T, then T is inferred as Type1

- T <:x Type2, then T is inferred as Type2

<T extends GetAreable> T findLargest(Seq<? extends T> seq)

- Shape o = A.findLargest(new Seq<Circle>(0));

- T <: Shape, GetAreable (i.e. T <: GetAreable), Circle <: T <: Object

**MY NOTES:**

**a.foo(d) where a, d are instances of D**

ctt of a is A, ctt of d is D (method would be chosen what is available for A)

method descriptor chosen is foo(A a)

rtt of a is D

method chosen is foo(A a) in B

(It will choose the nearest one as well for d.foo(d) since there was no (D d) method so it chose (C c))

**HEAP AND STACK:**

- the heap is the region in memory where all objects are allocated and stored, while the stack is the region where all variables are

allocated and stored.

-Stack: variables are contained within the call frames.

- Heap: The heap stores dynamically allocated objects  
- Class name, Instance fields and the respective values and Captured

Variables

**Immutability**

- Class and its fields are set as final

o Prevents assigning new value to the field, but does not prevent the

field from being mutated (final != immutable)  
o @SafeVarages annotation for passing an array of items ▪ (T …)

- Enabling safe concurrent execution

Nested Classes

**Static Nested Class:** Associated with the containing class

- Can only access static fields and static methods of containing class

**Non-static Nested Class (Inner Class):** Associated with the instance

- Can access all fields and methods of the containing class

**Local Classes:** Declared within functions

- declared in a block of code between { and }

- has access to variables of the enclosing class through the ‘this’

reference

- has access to local variables of enclosing method

**- Variable Capture**

o Local class makes a copy of local variables inside itself

**- Effectively Final**

o Local class can only access variables that are explicitly declared final

or implicitly final (effectively final)

▪ Variables cannot be re-assigned after initialisation

o Can be broken through reference type mutation

**Anonymous classes**: Declared and instantiated in a single statement

- has the format: new X (arguments) {body}

o X is the extended class / implemented interface

▪ Cannot be empty

▪ Cannot extend a class and implement an interface at the same

time, or implement multiple interfaces

o arguments are arguments to pass into the constructor

o body is the body of the class as per normal, except without a

constructor

- Same as local variables: captures the variables of the enclosing scope

(and must be effectively final)

**Side Effect-Free Programming**

**Pure Functions**: Does not cause any side effect

- Deterministic: Produces the same output, every single time

o Ensures referential transparency

- Functional Programming (FP) is building programs from pure functions

o OO languages utilise function-style programming

**Functional Interfaces**: Interfaces with a single abstract method

- @FunctionalInterface annotation

- No ambiguity about overridden methods

- Lambda functions can be used

o x -> x + 1

**Method Reference**

- Can be used to refer to

o static methods in classes

o instance method of a class / interface

o constructor of a class

- Box::of // x -> Box.of(x)

- Box::new // x -> new Box(x)

- x::compareTo // y -> x.compareTo(y)

- A::foo // (x, y) -> x.foo(y) or (x, y)

-> A.foo(x,y)

o Interpretation depends on how many parameters foo takes and

whether foo is a class method / instance method

o When compiling, Java searches for the matching method,

performing type inferences to find the matching method

▪ Compilation error thrown if multiple matches / ambiguity in

which method matches

**Curried Functions**: Translation of a general n-ary function to a sequence of n

unary functions

- Functions that take in multiple arguments

o e.g. X -> Y -> Z

- Allows for partial application of a function

**Lambda as Closure**

- “A lambda expression stores more than just the function to invoke -- it

also stores the data from the environment where it is defined. We call

such a construct that stores a function together with the enclosing

environment a closure.”

**Box and Maybe**

Lambda as a Cross-Barrier State Manipulator

- To preserve abstraction barrier, provide methods that accept a lambda

expression, apply it on the item, and return the new box with the new

value

Maybe: Option type (common abstraction)

**Lazy Evaluation**

Lazy Evaluation: Delay the execution of code until necessary

- Complex computations are built up without execution and only

evaluated on demand

Memoization: Caching the value to prevent re-computation

- Only for pure functions

**Infinite List**

- Computationally-efficient data structure that utilises lazy evaluation to

delay computation of data production

o Function that generates item (head)

o Function that generates an InfiniteList (tail)

- Further Improvements (InfiniteList<Lazy<Maybe<T>>>)

o Use Maybe<T> to encapsulate value of head

o Use Lazy<T> for memoisation

**Streams**

- Reduce (fold / accumulate): Applies a lambda repeatedly on elements to

reduce it into a single value

o **Takes in an identity value and accumulation function**

- Element matching: noneMatch/allMatch/anyMatch booleans

Intermediate Operations

- Operations that return another stream

o e.g. Stream::map, Stream::filter

- Stateful: Need to keep track of some states to operate

o e.g. sorted: returns a stream with the elements sorted

- Bounded: Should only be called on a finite stream

o e.g. sorted / distinct are bounded operations

- Truncation: Converting infinite stream to finite stream

- Peeking: Takes in a consumer, applying a lambda on a “fork” of the stream

o e.g. Stream.iterate(0, x -> x + 1).peek(System.out::println).takeWhile(x -> x <5). forEach(x -> {});

**Consumed Once**

- Streams can only be operated on once

- IllegalStateException will be thrown elsewise

**Loggable**

- Loggable class contains value and String log

- Loggable::flatMap makes Loggable general by abstracting out the operation

o e.g. Loggable.of(4).flatMap(x -> incrWithLog(x)).flatMap(x -> absWithLog(x))

o avoids the creation of fooWithLog for every function foo

- Can be made generic

**Monad**

- well-behaved programming constructs that have to follow three laws

Identity Law: Monad::of should behave like an identity

**- 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

**Associative Law:** Regardless of grouping of calls to flatMap, the behaviour

must remain 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)))

**Functors:** Common abstraction in functional-style programming

only ensures lambdas can be appliedsequentially to the value

- Abstraction that supports map

- Needs to adhere to two laws

**o Preserving identity:**

▪ functor.map(x -> x) is the same as functor

**o Preserving composition:**

▪ functor.map(x -> f(x)).map(x -> g(x)) is the

same as functor.map(x -> g(f(x))

- cs2030s.fp, Lazy<T>, Maybe<T>, and InfiniteList<T> are functors

**Parallel Streams**

**Concurrency:** Computation is divided into subtasks (threads)

- Allows the programmers to separate unrelated tasks into threads, and

write each thread separately

- Improves utilisation of the processor

o e.g. I/O in one thread, UI rendering in another, processor can

switch to rendering thread while waiting for I/O to complete

**Parallelism:** Multiple subtasks are running at the same time

- Processor is capable of running multiple instructions at the same time,

or multiple cores/processors and instructions are dispatched to them so

that they are executed at the same time

- All parallel programs are concurrent, but the converse is not true

**Parallel Stream:** adding the call parallel() into the stream

- embarrassingly parallel: Each elements is processed individually

without depending on other elements

o The only communication needed for each of the parallel subtasks is to combine the result of count()

o Lazy operation: can be inserted anywhere in pipeline after data

source and before terminal operation

**What can be parallelised?**

- Stream operations must not interfere with the stream data, and most of

the time must be stateless

o Side-effects should be kept to a minimum

- Interference: (applies for both stream() and parallelStream())

o One of the stream operations modifies the source of the stream

during the execution of the terminal operation

o ConcurrentModificationException thrown

- State vs Stateless: Whether the result depends on any state that might

change during the execution of the stream

o Stream.generate(scanner::nextInt) is stateful as it

depends on the state of the standard input

**- Side Effects:**

- list.parallelStream()

.forEach(x -> result.add(x));

▪ Generates a side effect (modifying result)

▪ ArrayList is what we call a non-thread-safe data structure. If

two threads manipulate it at the same time, an incorrect result may result.

o Three ways to resolve this .collect(Collectors.toList())

▪ new CopyOnWriteArrayList<>()

• Using a thread-safe data structure

▪ .toList() method that simply returns a list in the same orderas the stream

**- Associativity**

o Stream.of(1,2,3,4).reduce(1, (x, y) -> x \* y, (x, y) -> x \* y);

- reduce is inherently parallelisable as long as

▪ combiner.apply(identity, i) must be equal to i. **• i \* 1 equals i**

▪ The combiner and the accumulator must be associative -- the

order of applying must not matter. **• (x \* y) \* z equals x \* (y \* z)**

▪ The combiner and the accumulator must be compatible –

combiner.apply(u, accumulator.apply(identity, t)) must equal to

accumulator.apply(u, t) **• u \* (1 \* t) equals u \* t**

**Performance of Parallel Stream**: Parallelising a stream does not always

improve the performance. Creating a thread to run a task incurs some overhead

- overhead of creating too many threads might outweigh parallelisation benefits

- unordered() can be called as part of chain command to improve

efficiency provided respecting the original order is not important

**Threads**

**Synchronous Programming**: Method blocks until it returns java.lang.Thread

- Single flow of execution in a program

- constructor takes a Runnable instance as argument

- start() returns immediately

**Names**

- Every thread has a name

- getName() finds out the name of a thread

- Thread.currentThread() gets the reference of the current

running thread

**Sleep**

- causes current execution thread to pause execution immediately for a

given period

\*isAlive() checks if another thread is still running

**Asynchronous Programming**

Limitations of Thread

**- Coordination**:

o There are no methods that return a value

▪ Threads communicate through shared variables

o No mechanism to specify execution order / dependencies

▪ which thread to start after another thread completes

o Possibility of exceptions in each of the tasks

**- Overhead:**

o Creation of Thread instances takes up resources

▪ They should be reused to run multiple tasks

o Managing the instances itself and deciding which instance should

run which Thread is difficult

**CompletableFuture Monad**

- allows tasks to be performed concurrently (encapsulation of promise)

- **Creation:**

o completedFuture: already completed task

o runAsync: Accepts a Runnable lambda expression

▪ Completes when the given lambda expression finishes

o supplyAsync: Accepts a Supplier<T>

▪ Completes when the given lambda expression finishes

Chaining: \* Asynchronous versions exist as well (e.g. thenApplyAsync)

o thenApply: map

o thenCompose: flatMap

o thenCombine: combine

o thenRun: executes Runnable after current stage completes

o runAfterBoth: executes after current stage and input CF completes

o runAfterEither: executes after either completes

- **Getting the Result:**

o get() blocks until CompletableFuture completes (synchronous call)

▪ InterruptedException: thread as been interrupted

▪ ExecutionException: errors/exceptions during execution

o join(): same as get() except without any checked exception thrown

- Exception Handling:

o others methods of handling: exceptionally, whenComplete

o handle: handles exception (takes BiFunction)

▪ BiFunction(value, exception, return value)

▪ .handle((t, e) -> (e == null) ? t : 0)

**Fork and Join**

**Thread Pool**

- Consists of:

o Collection of threads, each waiting for a task to execute

o Collection of tasks to be executed

- Tasks are put in a shared queue, and an idle thread picks up a task from

the shared queue to execute

**Fork and Join**

- ForkJoinPool: fine-tuned for the fork-join model of recursive parallel execution

- parallel divide-and-conquer model of computation

- solve a problem by breaking it up into smaller (but identical) problems,

and then combining the results

- RecursiveTask<T> supports methods fork() and join(), as well as compute()

o left.fork(): add tasks to thread pool (one of the threads will

call its compute() method)

o right.compute(): normal method call

o left.join(): blocks until computation of recursive sum is completed and returned

**ForkJoinPool**

- Each thread has a deque of tasks

o deque is a double-ended queue, and behaves like both stack and queue

- When a thread is idle, it checks its deque

o If deque is empty, it picks up a task at the head of the deque

o If deque is empty, it picks up a task from the tail of the deque of another thread to run (work stealing)

- When fork() is called, the caller adds itself to the head of the deque

of the executing thread

o Most recently forked task gets executed next (similar to how

normal recursive calls)

- When join() is called

o If subtask to be joined hasn’t been executed, compute() method is

called and subtask is executed

o If subtask to be joined has been completed (by this thread or stolen

and completed by another thread), the result is read and join() returns

o If subtask to be joined has been stolen and is being executed, the

current thread finds some other tasks to work on either in its local

deque or steals another task from another deque

**Order of fork() and join()**

- Most recently forked task is likely to be executed next

o join() the most recent fork() task first

- Order of forking should be the reverse of the order of joining

o left.fork();

right.fork();

return right.join() + left.join();

- Should only be at most a single compute (in the middle of the

palindrome)

**MY NOTES:**  
-Final and static methods cannot be overridden.

-The overriding method must have same argument list.

-The overriding method must have same return type (or subtype).

-The overriding method must not have more restrictive access modifier.

-The overriding method must not throw new or broader checked exceptions.

-**MONADS**

- m.map(x -> f(x)).map(x -> g(x)) 🡺

m.flatMap(x -> Monad.of(f(x))).flatMap(x -> Monad.of(g(x))) 🡺

m.flatMap(x -> Monad.of(f(x)).flatMap(x -> Monad.of(g(x)))) (ASSC. LAW) m.flatMap(x -> Monad.of(g(f(x)))) (LEFT IDENTITY LAW) 🡺

m.flatMap(x -> Monad.of(g(f(x))))

-A a = new A(); Producer p = (Producer) a;

-Java allows type casting here because there is a possibility that during run-time, a is aninstanceofaclassthatextends A andimplementsthe Producer interface.

-Producer<Integer> p = (Producer<Integer>) a; [WARNING]  
During run-time,there is no way for the type checker to check that a is an instance of Producer<Integer> since the type argument Integer has been erased.

- A is a final class

-It is not possible that during run-time, a is an instance of a class that extends from A , as A is declared as final. So the run-time type of a must be A and A does not implement Producer .

**STACK HEAP😐**

A diagram of a diagram

Description automatically generated

class B {

static int x = 0;

void f() {

A a = new A(); }

static class A {

int y = 0;

A() {

this.y = B.x + 1; // <-- } }}

A diagram of a diagram

Description automatically generated

class B { int x = 1;

void f() { int y = 2;

class A {   
void g() {

x = y; } }

A a = new A(); // Line A a.g();} }

A diagram of a diagram

Description automatically generated

class A {

private int x; public A(int x) {

this.x = x; }

public int get() { // Line A

return this.x;} }

A a = new A(5);

Producer<Integer> p = () ­> a.get(); p.produce();

- Stream<Long> Omega (int n) {

return IntStream. range (1, n) -mapToObj(x -> Long. valueOf(x)); }

-ff -> x -> g.transform(ff.transform(x)) (CURRYING)

- 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))

); }

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

}  
  
Monad.of(x)

.flatMap(i -> Monad.of(i).flatMap(i -> em(i)).flatMap(i -> fm(i)))

.flatMap(i -> Monad.of(i).flatMap(i -> gm(i)).flatMap(i -> hm(i)))  
=> by left identity

Monad.of(x)

.flatMap(i -> em(i).flatMap(i -> fm(i)))

.flatMap(i -> gm(i).flatMap(i -> hm(i)))

=> by associativity

Monad.of(x)

.flatMap(i -> em(i)).flatMap(i -> fm(i))

.flatMap(i -> gm(i)).flatMap(i -> hm(i))