Programs		
Compiled	Translated to Machine Code	Javac (.class)
Interpreted	Codes directly Read and Executed	Java to (.java)

Programming Languages: Typed Property		
Dynamic	Variable can hold values of different unrelated types	Python, Javascript
Static	Variable types are declared, and only hold values of that type and subtypes	Java, C
Strong	Enforce strict rules in type system, ensuring type safety (catch during compile time)	Java
Weak	Allow typecasting that changes interpretation of byte	C
Primitive	Predetermined values of the language, never sharing value with each other (byte, short, int, long, float double, char)	
Reference	Points/References to object instances	

Java Primitive Type Sizes (bits)					
boolean	1	char	16	byte	8
short	16	int	32	long	64
float	32	double	64		
$byte <: short <: int <: long <: float <: double$					
$char <: int$					

Subtype Properties	
Reflexive	For any type S, we have $S <: S$
Transitive	$(S <: T) \wedge (T <: U) \rightarrow S <: U$
Anti-Symmetric	$(S <: T) \wedge (T <: S) \rightarrow S = T$

Object-Oriented Programming Principles	
Abstraction	Hides internal details (Method, Variable Names) Composite Data Type (Struct, Class, Object)
Encapsulation	Bundles Data and Methods in Class (Private Data, Public Method)
Inheritance	IS-A relationship, extends the parent class, sharing a set of properties
Polymorphism	Refer to "Dynamic Binding" table, allow Override, Specifically, Inclusion Polymorphism

OOP Principles Implications	
Special Reference Value: Null	Variable uninitialized will take the value: null Refers to a non-existent instance
Keywords	Refer to "Keywords" table, determines characteristics of the Class, Method or Field
Abstraction Barrier	Implementer: Implements Codes Client: Use Codes (No idea about implementation)
Reduced Code Complexity	Functions group a set of actions and codes, hiding implementation, simplifying the code, reduce repetition
Data Hiding	Private: Accessible only withing Class

	Public: Accessible in and outside of Class
Tell, Don't Ask	Refer to "Tell, Don't Ask" table
Class, Fields, Methods, Interface	Refer to the "Class, Fields, Methods, Interface" table
Composition	A Class as a Class Field (ie Circle has Point) Objects can be shared (Hence Accessors for Objects can be dangerous), HAS-A Relationship
Run-Time Type	The exact type of the object the variable points to, must be a Subtype of Compile Type
Liskov Substitution Principle (LSP)	When overriding, maintain the spirit of the method, "Let ϕ(x) be a property provable about objects x of type T. Then ϕ(y) should be true for objects y of type S where S<:T."
LSP in Testing Context	A subclass should not break the expectations set by the superclass If class B is substitutable for a parent class A then it should be able to pass all test cases of the parent class A. If it does not, then it is not substitutable and the LSP is violated
Type Conversion	Narrowing: S to T where S <: T, lose information Widening: T to S where S <: T

Keywords	
Access Modifiers	*default*: Accessible within the package public, private, protected
public	Accessible to all classes
private	Inaccessible to all classes
protected	*default* + accessible in child class
this	Points to the instance / object itself, Only relevant to initialised instances ie cannot access static value
super	Points to the instance immediate parent
static	Makes variable / method a Class Method (ie shared in Class)
@Override	Explicitly declares overriding method of a parent, throws "OverrideError" if parent method not found
final	Fields: Can't Re-Assigned Method: Can't be Overriden Class: Can't be inherited
abstract (Concrete: !abstract)	Class: Can contain abstract method, cannot be instantiated Methods: No body declared, Must be declared in concrete subclasses
try catch (E e) finally	try: start block, stops upon Exception thrown to catch catch (E e): runs block if Exception thrown in try matches E finally: runs block after try or catch, executed even after return or throw is called
@Suppress Warnings("")	Tells compiler to ignore warnings, used with String arguments "rawtypes" or "unchecked" to ignore the respective warnings. Good practice to add comments as to why the warnings are suppressed ie, why it is safe
@SafeVarags	Tells compiler that the generic arguments are safe and ignore unchecked warning

Tell, Don't Ask	
Description	Tell Class to do the work, not ask for the data and manipulate them
Accessors	Getter Methods: returns the data
Mutators	Setter Methods: changes the data
DANGER	Able to edit the values (without verification) Applicable to Accessors when data is an Object

Class, Fields, Methods, Interface	
Class & Interface Declaration Order	Public/private? static? final? name (extends Class) (implement Interface)
Fields & Methods Declaration Order	public/private? static? final? return name
void Return Type	Special Type of Methods to return nothing
Interface	Collection of implicitly "public abstract" Methods
Static Class Field	A field shared within a class: Belongs to the Class (Different from Static Typed: Property of Programming Language)
Non-Static Class Field	Fields only initialised and accessible from the instance of the class holding it: Belongs to the Instance
Static Class Methods	A method shared within a class: Belongs to the Class, Unable to access Non-Static Class Fields
Non-Static Methods	A method only initialised and accessible from the stance of the class holding it: Belongs to the Instance, Able to access Static and Non-Static Class Fields

main method	
Descriptions	Entry point to the program
Declaration	public static final void main(String[] args) {}

Heap & Stack (By Java Virtual Machine, JVM)	
Method Area	Stores code for the methods
Metaspace	Stores meta information about classes
Heap	Stores Dynamically Allocated Objects
Stack	For local variables and call frames Last-In-First-Out (LIFO)
Empty ∅ (null)	Denote uninitialized variables
Pointers	Points to the Objects in Heap, Primitives are stored directly to the variables
Garbage Collector	Checks for unreferenced objects on heap and cleans up the memory automatically
Aliasing	2 Pointers to the same object

Object Class	
Immutable Objects	Its state cannot change after construction
Implicit Inheritance	Classes that do not extend another class inherits from Object implicitly

toString() Method	Converts reference object to a String object, called implicitly by Java
equals(Object obj) Method	Check if Object input refers to the same Object instance
@Override	Override parent method with the same method descriptor + return subtyped
Method Signature	Method Name; number, type and order of Parameters
Method Descriptor	Method Signature + Return Type
Method Overloading	Methods with same name, differing Method Signature

Wrapper Classes					
Wrapper Class		Encapsulate a primitive type, Immutable			
Auto-boxing & Unboxing		Auto-Boxing: primitive to Wrapper Unboxing: Wrapper to primitive			
Tradeoff vs Primitive		Performance & Memory Allocation			
byte	Byte	short	Short	int	Integer
long	Long	float	Float	double	Double
char	Character	boolean	Boolean		

Dynamic Binding (Late Binding / Dynamic Dispatch)	
Description	Method of same signature invoked is decided based on run-time type of instance calling the method
Method Invocation	Method Descriptor: Compile Time (Target Type, Param Number, Type, Order) Method Implementation: Run-Time (Target Type)
Example	Given Class A, with methods equals(A), override equals(Object): Object.equals(Object / A)⇒Object::equals(Object) (Object obj = A).equals(Object / A)⇒A::equals(Object) A.equals(Object obj = Object / A)⇒A::equals(Object) A.equals(A)⇒A::equals(A)
Class Method Invocation	Does not support Dynamic Binding, resolved statically during compile time

Type Casting	
Description	Ask compiler to trust that instance has a run-time type of a subtype
Relationship	At Compile Time: must have subtype relationship ie (S) T, then S<:T or T<:S At Run Time: Referenced Instance must be subtype of casting type
Run-Time Class Mismatch	When the Run Time relationship stated above is not met
Casting to Interface	If undeclared, assumes child class (if not final) may implement interface so no error/warning thrown gives uncheckedCastWarning if interface is generic since unable to check after type erasure

Variances	
Covariant	$S <: T \Rightarrow C(S) <: C(T)$
Contravariant	$S <: T \Rightarrow C(T) <: C(S)$
Invariant	Neither Covariant nor Contravariant
Example	Array: Covariant (Integer <: Object⇒Integer[] <: Object[]) Generics: Invariant (Seq<Integer> not a variant of Seq<Object>, vice versa) Upper Bounded Wildcards: Covariant (Seq<? extends Integer> <: Seq<? extends Object>) Lower Bounded Wildcards: Contravariant (Seq<? super Object> <: Seq<? super Integer>)

Exception	
Exception	Subclass of Throwable Errors managed with try / catch / finally
Unchecked Exception	Caused by programmer's error, subclass of RuntimeException Eg. IllegalArgumentException, NullPointerException, ClassCastException
Checked Exception	Out of programmer's error, user error, Must be handled or cannot compile Eg. FileNotFoundException, InputMismatchException
Method Throwing Exception	<method descriptor> throws Exception { throw new Exception(message?); }
Catch Exceptions to Clean Up	Handle the Exception appropriately based on the Exceptions caught
BAD: Pokemon Exception Handling	Catching all Exceptions ie catch (Exception e)
BAD: Overreacting	Exiting program when Exception thrown, prevents calling function from cleaning up resources, worse, exiting program silently ie without comment
BAD: Breaking Abstraction Barrier	Leaking information of implementation behind abstraction barrier
BAD: Use Exception as Control Flow Mechanism	Intentionally throwing an Exception in try block to go to a catch block, may end up catching a valid but unintended Exception
Error Class	For situations where program should terminate as generally no way to recover from error, typically no need to create or handle such errors Eg. OutOfMemoryError, StackOverflowError

Generics	
Generic Types	Takes on other types as type parameters Eg. <T>
Type Arguments	To put into type parameters during instantiation Eg. <String>, <S>, <>
Parameterized Type	Instantiated Generic Type
Generic Classes	Eg. Pair<S, T>, Array<T>
Generic Methods	Declare generic type before return type, parameter type is scoped within the whole method Eg. <T> T getFirstElem(T[] tArr) { return tArr[0]; }

Bounded Type Parameters	Sets boundary of generic type with extends or super
Type Erasure (Generic Class)	Due to code sharing approach of Java, Java erases type parameters and type arguments during compilation. Transform Generic Classes or Methods to type parameters upper bound Eg. Pair<String, Integer>("", 1).getFirst() to (String) Pair("", 1).getFirst()
Bridge Methods (Parameterised Class)	Compiler silently create bridge methods with the same name but different signature so that it matches with parent generic class / Parameterized class will inherit generic class methods, so compiler bridge the inherited method to the parameterized method, thus allowing polymorphism, thus allowing polymorphism Eg. Pair<String, Integer>("", 1).getFirst() to (String) Pair("", 1).getFirst()
Type Erasure & Bridge Method Example	Eg A<T>::set(T t), B<:A<String>, B::set(String s) Type Erasure: A<Object>::set(Object o) B::set(String s) does not override A::set(Object o) Bridge Method: B::set(Object o) {B::set((String) o)}
Generics & Arrays	Generics and Arrays can't mix, Arrays are reifiable, but Generics are non-reifiable due to type erasure Eg. new Pair<String, Integer>[int] to new Pair[int]
Rule of generic array	Generic array *declaration* is fine but generic array *instantiation* is not Eg. T[] arr
Heap Pollution	A term that refers to the situation where a variable of a parameterized type refers to an object that is not of that parameterized type
Reifiable Type	A type where full type information is available during run-time
Seq Class	Wrapper class for array to allow safer type erasure
Raw Type	A generic type used without type arguments Eg. Seq
instanceof	Checks type of instance Eg. String instanceof Object
Suppress Warnings	unchecked: Compiler unable to guarantee type erasure is safe rawtype: Use of rawtypes, can refer to any instance of any type
-Xlint:	Use with unchecked / rawtype to get warning message
Wildcards	Denoted as ?, can be used as a substitute for any type, Can be interpreted as a set of any type
Upper Bounded Wildcards	Denoted as <? extends Class>, the wildcard will only accept substitutes for subclasses of Class, Can be interpreted as a set of types that are subclasses of Class, is covariant
Lower Bounded Wildcards	Denoted as <? super Class>, the wildcard will only accept substitutes for superclasses of Class, Can be interpreted as a set of types that are subclasses of Class, is contravariant
Unbounded Wildcards	Denoted as <?>, is supertype of every parameterized type of its class, allow flexibility for methods to accept all types, An appropriate substitute for Rawtypes

	Eg. Class<AnyType> <: Class<?>
PECS	Short for "Producer extends, Consumer super", denotes the generic boundaries for the parameters of Producer (produces value) or Consumer (consumes / takes in value) classes

Type Inference	
Description	Decides what type the output will be
Target Typing	Return type must be subtype of the target's type
Type Bounds	Method Type: Generic Type in diamond operator <> Return Type: Generic Type returned by Method Argument Type: Generic Types in Method Argument
Considerations	Given a Type range, pick most specific type that satisfies all types in the bound range
Examples	Type1 <: T <: Type2, pick T = Type1 Type1 <: T, pick T = Type1 T <: Type2, pick T = Type2

Immutable Classes	
Description	No changes can be made to instances of Immutable classes, enabling Safe Sharing of Objects & Internal
Problem to Solve	When two instances of a separate class share the same instance of a mutable field, a modification to the mutable field instance will affect both composing instances.
Implementation Must (Should) - Have	Declare the immutable class as final to disallow inheritance to avoid mutable subclass. Ensure fields are immutable.
Implementation #1	Declare all fields as final
Implementation #2	Share copies of the field information for getter methods or modifications ie clone() method

Variadic Method	
Varargs	Appears as T... in method argument Syntactic sugar for passing an array of items to a method
Variadic Method	Method with a variable number of arguments, ie contains a varargs in argument
@SafeVarargs	Tells compiler to ignore unchecked warning for generic varargs

Nested /Local Class	
Nested Class Description	Declared within a container class, tends to be used as a "helper" class that serve specific purposes
Local Class Description	Declared within a method, scoped / exists within the method
Characteristics	Able to access field and methods of container class including private
Implementation	Declare as private as typically not exposed to the client outside of abstraction barrier

	Should have the same encapsulation of container class as container class may leak implementation details to the nested class
Static nested class	Associated with containing class, not an instance, thus can only access static fields and methods of containing class
Non-static nested class ie inner class	Able to access all fields and methods of containing class
Qualified this	Helps nested class point to a field or method of the container class ie ContainerClass.this
Private nested class	Cannot be interacted with outside of container class ie No type assigning, constructor, method calls, field access outside of container class
Variable Capture (Local Class)	When a method returns, all local variables are removed from stack. Hence the local class makes a copy of local variables inside itself. Local variables must be final or implicitly final
Effectively Final Variables	Local variables cannot be reassigned; however, reference types may be mutated
Anonymous Class	Syntactic sugar to declare a local class without assigning the class a name

Functional Programming / Side Effect-Free Programming	
Pure Function Description	Treating methods as mathematical function, takes in an input and produce an output
PF Characteristic	No side-effects ie no print, write, assign, exception Deterministic: same input gives same output, ensures referential transparency
PF Application	Used in immutable classes
Functional Interface	An interface with exactly one abstract method Annotated with @FunctionalInterface No ambiguity about which method overridden
Lambda Expression (LE*)	Replaces functional interface boilerplate General form: (param) -> {body}; Single Return Statement: (param) -> body;
Method Reference	In the form Class/Instance::Method, can refer to: Static method in a class Instance method of a class or interface Constructor of a class *A::foo can be a -> a.foo() or a -> A.foo(a), determined by the actual input and if foo() is a class or instance method
Curried Function	Function that returns a function ie n-ary function to a sequence of n unary function ie (x, y) -> f(x, y) to x -> y -> f(x, y)
Lambda as Closure	LE* also stores the data from the environment where it is defined, ie localClass::Method
Lambda as a Cross-Barrier State Manipulator	Instead of providing setter or getter methods which can be abused, methods can be passed to the class to manipulate internals without exposing them, allowing the class to handle the semantics

Eg. Maybe, Lazy, InfiniteList	See: map, flatMap
Lambda as Delayed Data	Store expressions to execute them later, lazy evaluation where we only execute when we need to. See: Producer, Task
Eager Evaluation	Evaluate immediately, opposite of Lazy Evaluation
Memoization Eg. Lazy	Since we produce an object when putting an input and calling its getter produces the same output, we can store the output, so we only evaluate once
Infinite List	Our implementation of Stream

Java Implementations		
Classes / Functional Interfaces	CS2030S	java.util.function
	BooleanCondition<T>::test	Predicate<T>:test
	Producer<T>::produce	Supplier
	Consumer<T>::consume	Consumer<T>::accept
	Transformer<T, R>::transform	Function<T, R>::apply
	Transformer<T, T>::transform	UnaryOp<T>::apply
	Combiner<S, T, R>::combine	BiFunction<S, T, R>::apply
	Box<T>	N/A
	Maybe<T>	java.util.Optional<T>
	Lazy<T>	N/A
	InfiniteList<T>	java.util.stream.Stream<T>

Stream	
Description	An InfiniteList with more functionalities
Bonus	Arrays::stream and List::stream exists
Terminal Operations	Triggers the evaluation of the stream, is Eager Eg. forEach, reduce
Consumed Once	Stream can only be operated once
Intermediate Stream Operations	Returns another stream with operated elements, are Lazy and does not cause the stream to evaluate Eg. map, filter, flatMap
stream::flatMap	Transforms every element in the stream into another stream, and the resulting stream of streams is then flattened and concatenated
Stateful Operations	Need to keep track of some states to operate Eg. sorted, distinct
Bound Operations	Should only be called on a finite stream Eg. sorted, distinct
Truncating stream	Converts infinite stream to finite stream Eg. limit, takeWhile
stream::peek	Takes a Consumer, applying a LE* on a "fork" of the stream

stream::reduce	Reduce the stream into a value
	Pseudocode:
	result = identity;
	for each element in stream:
	  result = accumulator.apply(result, element);
	return result;

Element Matching	Tests if elements in a stream pass a given predicate

	Eg. noneMatch, allMatch, anyMatch

Monad	
Description	"Well behaved" classes following three laws
Identity Law	Monad::of should produce an identity Monad::flatMap should simply apply the given LE* to the value Monad:: of and Monad::flatMap should do nothing extra to the value and side info
Left Identity Law	Monad.of(x).flatMap(x -> f(x)) == f(x)
Right Identity Law	monad.flatMap(x -> Monad.of(x)) == monad
Associative Law	monad.flatMap(x -> f(x)).flatMap(x -> g(x)) == monad.flatMap(x -> f(x).flatMap(y -> g(y)))

Functor	
Description	Simpler construction than monad, only ensure lambdas can be applied sequentially to the value
Preserves Identity	functor.map(x -> x) == functor
Preserves Composition	functor.map(x -> f(x)).map(x -> g(x)) == functor.map(x -> g(f(x)))

Parallel Stream	
Sequential	Complete one task at a time
Concurrency	Work on multiple threads, one instruction at a time
Parallelism	Work on multiple threads, multiple instruction at a time, eg all parallel prog are concurrent but not all concurrent prog are parallel
stream::parallel	Stream is broken down into subsequences and operations are applied for each subsequences, thus the output will be unordered
stream::sequential	Stream converted to sequential, latest call triump
Parallel Conditions	Stateless, no side effects
Interference	Stream operation modifies the source of the stream during execution of terminal operation
Stateful	Result depends on any state that might change during execution of the stream. Eg. generate, map
Side Effects	Something else is affected during execution
Non-Thread-Safety	When two threads manipulate a non-thread-safe data structure, it may produce an incorrect result
Thread-Safety	1) Use Stream::collect(Collectors.toList()) 2) Use thread-safe data structure ie java.util.concurrent.CopyOnWriteArrayList 3) Use Stream::toList (in Java 21)
Parallel Reduce	Using reduce(identity, accumulator, combiner) Accumulator: accumulates the sub-streams Combiner: combines accumulator results
Parallel Reduce Identity Rule	Applies to combiner combiner.apply(identity, i) == i
Parallel Reduce Associative Rule	Combiner and accumulator must be associative The order of applying must not matter
Parallel Reduce Compatible Rule	Combiner and accumulator must be compatible combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)
Parallel Performance	Parallelizing stream doesn't always improve performance since creating thread to run task incurs overhead which may outweigh the benefit
Ordered vs Unordered	Ordered: defined encounter order, ie streams from iterate and ordered collections (List, array) Unordered: streams from generate or unordered collections (Set)
Ordered Operations	Some operations respect the encounter order ie preserves the original order eg distinct, sorted / coordinate between streams to maintain order eg findFirst, limit, skip, takeWhile These makes parallelizing stream expensive
Unordered stream	Given an ordered stream and respecting the original order is not important, can use Stream::unordered to make parallel operations much more efficient

Asynchronous Programming	
Synchronous Programming	Method blocks the program flow ie program waits for the return of the method
Asynchronous Programming	Method returns an object immediately that can be tracked for the progress or completion of the encapsulated function
java.lang.Thread	Used to encapsulate a function to run in a separate thread
Thread::start	Returns immediately, does not wait until the thread's encapsulated function is done
Thread::getName	Get name of the target thread
Thread:: currentThread	Get the reference of the current running thread
Thread::sleep	Cause the current thread to pause execution immediately for a given period (in ms)
Thread::isAlive	Checks if another thread is still running
Limitation of Thread	Requires careful coordination No methods in Thread return a value No mechanism to specify execution order and dependencies among them Overhead: Creation of Thread instances takes up some resources in Java
Note*	Program exits only after all the threads created run to their completion

CompletableFuture	
java.util.concurrent. CompletableFuture <T> CF*	A monad that allows us to perform the task concurrently, encapsulates the promise to produce a value
Promise	Encapsulates a value that is either there or not there yet

cf*::get	Waits for all concurrent tasks to complete and return us a value, throws InterruptedException and ExecutionException to be caught and handled
cf*::join	Same as cf*::get but no checked exception is thrown
cf*::isDone	Returns if the CF* instance completed in any fashion: normally, exceptionally, or via cancellation
CF*:: completedFuture	Creates a CF* that is already completed and ready to return the value
CF*::runAsync	Takes in a Runnable LE*, return type CF*<Void>, completes when the LE* finish, runs the CF* asynchronously immediately
CF*::supplyAsync	Takes in a Supplier<T> LE*, return type CF*<T>, completes when the LE* finish
CF*::allOf/anyOf	Returns a CF* that completes when all/any supplied CF*s are completed
cf*::thenApply/ Compose/Combine	Analogous to map, flatMap, combine respectively
cf*::then...Async	Given LE* is run on a different thread
cf*::thenRun	Takes in a Runnable and executes it after the target CF is completed, returns CF*<Void>
cf*::runAfterBoth/ runAfterEither	Takes in another cf* and Runnable, executes after the Runnable and/or input cf is completed, returns CF*<Void>
cf*::runAfter...Async	Similar to cf*::then...Async
cf*::exceptionally	Takes in a Function<Throwable, T> which will only executed if an exception is thrown during execution, returns a CF*<T> that contains the result of either normal or exceptional execution
cf*::whenComplete	Takes in a BiConsumer<T, Throwable> that executes when the target cf* completes, returns the CF*<T> with the result or exception from the BiConsumer
cf*::handle	Takes in a BiFunction<T, Throwable, U> that executes at cf* completion, T = null @ exceptional, Throwable = null @ normal, returns the CF*<U> with the result or exception

ForkJoinPool	
java.util.concurrent. ForkJoinPool<V>	Java thread pool implementation for the fork-join model of recursive parallel execution
Fork-join model	Essentially parallel divide-and-conquer model, splitting a task to smaller size (fork) recursively and then combining them (join)
java.util.concurrent. RecursiveTask<V> RT*	Abstract class that supports fork and join methods
rt*::fork	Submits smaller version of the task for execution
rt*::join	Waits for smaller tasks to complete and return
rt*::compute	Abstract method to define what the task should do
How ForkJoinPool Works	Each Thread has a deque of tasks to execute If thread is idle, checks its deque and do:

	If deque not empty, execute head of dequeue Else, work steal from another thread If rt*::fork called, rt adds itself to the head of executing thread dequeue If rt*::join called, do: If rt not executed, call rt::compute Else If rt completed, return result Else ie stolen and executing, idle until result
Work Steal	Executes tail of another thread's dequeue
Order of fork, compute, join	Should form a palindrome with no crossing, at most 1 compute at the middle of the palindrome