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

Java Primitive Types Relationship
$byte <: short <: int <: long <: float <: double, char <: int$

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

OOP Principles Implications	
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 code, reduce repetition
Data Hiding	Private: Accessible only withing Class Public: Accessible in and outside of Class
Tell, Don't Ask	Tell Class to do the work, not ask for the data and manipulate them
Composition	A Class as a Class Field
Liskov Substitution Principle (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	Widening: S to T where S <: T, lose information Narrowing: T to S where S <: T

Keywords	
Access Modifiers	*default*: Accessible within the package public: Accessible to all classes private: Inaccessible outside of the class protected: *default* + accessible in child class
@Override	Override parent method with the same method descriptor + return subtype
final	Can't Re-Assigned, Overridden, Inherited
Try, catch (E e), finally	finally: runs block after try or catch, executed even after return or throw is called

@SafeVarags	Tells compiler that the generic arguments are safe and ignore unchecked warning

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
Interface	Collection of implicitly "public abstract" Methods
Static Methods	Unable to access Non-Static Class Fields
Non-Static Methods	Able to access Static and Non-Static Class Fields

Heap & Stack (By Java Virtual Machine, JVM)	
Method Area	Stores code for the methods
Metaspace	Stores meta information about classes ie static
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	
Method Signature	Method Name; number, type and order of Params
Method Descriptor	Method Signature + Return Type
Method Overload	Methods with same name, different signature

Wrapper Classes	
Wrapper Class	Encapsulate a primitive type, Immutable
Auto-boxing & Unboxing	Auto-Boxing: primitive to Wrapper Unboxing: Wrapper to primitive

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	Compile Time: Method Descriptor Run Time: Actual object type
Static Method Invocation	Does not support Dynamic Binding, resolved statically during compile time

Type Casting	
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
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

Exception	
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 with try catch or cannot compile Eg. FileNotFoundException, InputMismatchException
Bad Exception Handlings	Pokemon: catching all exception Overreacting: overcompensating / exit program Breaking Abstraction Barrier: Reveal internal info Exception as control flow: exception as "if" alt

Generics	
Type Erasure (Generic Class)	Transform Generic Classes or Methods to type parameters upper bound for Generic Class
Bridge Methods (Parameterised Class)	Parameterized class will inherit generic class methods, so compiler bridge the inherited method to the parameterized method

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
Reifiable Type	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
Suppress Warnings	unchecked: can't guarantee type erasure is safe rawtype: Use of rawtypes
Wildcards	Denoted as ?, can be used as a substitute for any type, Can be interpreted as a set of any type
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<?>

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

Immutable Classes	
Description	No changes can be made to instances of Immutable classes, enabling Safe Sharing of Objects & Internal

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

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
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
Functional Interface	An interface with exactly one abstract method
Method Reference	*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	(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: map, flatMap

Lambda as Delayed Data	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

java.util.stream.Stream<T>	
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, leave stream unevaluated Eg filter, map

Monad	
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	
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	
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
Parallel Conditions	Stateless, no side effects
Non-Thread-Safety	When two threads manipulate a non-thread-safe data structure, it may produce an incorrect result
Thread-Safety Implementation	1) Use Stream::collect(Collectors.toList()) 2) Use Stream::toList (in Java 21)
Parallel Reduce	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)
Ordered vs Unordered	Ordered: defined encounter order, ie stream::iterate Unordered: stream::generate, Set
Ordered Operations	Some operations respect the encounter order Eg distinct, sorted / coordinate between streams to maintain order Eg findFirst, limit, skip, takeWhile

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	
Asynchronous Programming	Method returns an object immediately that can be tracked for the progress or completion of the encapsulated function
Thread: Note*	Program exits only after all the threads created run to their completion

java.util.concurrent.CompletableFuture<T>	
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, no checked exception is thrown
cf*::isDone	Returns if the CF* instance completed in any fashion: normally, exceptionally, or via cancellation
CF*::runAsync	Takes in Runnable, runs the Runnable asynca
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…	Starts after target CF* is done
cf*::…Async	Given LE* is run on a different thread
cf*::handle	Takes in (T t, E e) -> { return U u}

java.util.concurrent.ForkJoinPool, java.util.concurrent.RecursiveTask<T>	
Fork-join model	Essentially parallel divide-and-conquer model, splitting a task to smaller size (fork) recursively and then combining them (join)
RecursiveTask<T>	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