Types

S is a *subtype* of T (`S <: T`) if a piece of code written for variables of S can be used for variables of T

1. **Reflexive** - $S <: S$
2. **Transitive** - if $S <: T$ and $T <: U$, then $S <: U$
3. **Anti-symmetry** - if $S <: T$ and $T <: S$, then $S = T$

`S instanceof T` returns **true** if $S <: T$

Everything (incl. interfaces) is a subtype of `Object`

Type Conversions

Narrowing	Widening
$S <: T$, we can type cast a variable of type T to type S	$S <: T$, we can type cast a variable of type S to type T
`Shape s = new Circle()` `Circle c = (Circle) s`	`Shape s = new Circle()`

Primitive Types

byte → short → int → long → float → double
char → int → long → float → double

Variance of Types

$C(S)$ stands for complex types - arrays
Covariant - $S <: T$ implies $C(S) <: C(T)$
Contravariant - $S <: T$ implies $C(T) <: C(S)$
Invariant - neither *covariant* nor *contravariant*

* Java **array** is *covariant* / Java **generics** are *invariant*
* Java arrays and generics **cannot be mixed**

Run-time VS Compile-Time

`Colour c = new Red()` ⇒ **CTT** - `Colour` / **RTT** - `Red`

OOP Principles

Information Hiding - `private` fields, `public` methods
Tell-Don't-Ask - client should not perform computation

Encapsulation

Composition - *HAS-A* relations / **Inheritance** - `extends` - *IS-A* relationship

Polymorphism

Dynamic Binding
1. Determine compile-time type of target
2. Look for all available methods
3. Choose the most specific method
4. Determine run-time type of target

Method Signatures and Descriptors

Method Signature - method name, number of parameters, type of each parameter, order of parameters - `C::foo(B1, B2)`

Method Descriptor - method signature + return type - `A C::foo(B1, B2)`

Method Overriding and Overloading

Overriding - Same method descriptor (**subtypes** of return type allowed)
Overloading - Same method name, but different method signatures

Liskov Substitution Principle

If $S <: T$:

- S should be able to pass all test cases of T
- S should not break any expectations and expected attributes of T - i.e. S should contain all fields and methods of T
- S should be able to replace T and not break any logic
- inputs of S must be **inclusive** of inputs of T, output of S must be a **subset** of T

Preventing Inheritance (classes) and Overriding (methods) - `final`

Interfaces

`interface GetAreable` - Models what an entity can do

`class A implements X, Y, Z` - $A <: X$, $A <: Y$, $A <: Z$

`interface X extends Y, Z` - Interfaces can extend multiple interfaces, but **cannot implement** (interfaces are abstract!)

`interface C , class D , D d = new D()`
`C c = (C) d` compiles but subjected to runtime check since `d` may be part of a future class that is a subtype of both `C` and `D` (e.g. `class E extends D implements C` where `D d = new E()`)

Exceptions

Overridden methods in subclasses can throw the same exceptions or any of its subtypes (not throwing is also fine)

```java
class NewCheckedExceptions extends Exception {
    public NewCheckedExceptions() { super("msg") }}
class C throws NewCheckedException {
        throw new NewCheckedException("message") }
...
try { // do something }
catch (NewCheckedException e | exception 2) {
    System.out.println(e.getMessage());
    // handle exception 2 }
finally { // runs regardless of exception }
```

Generics

- Enforces **type safety** → binds a generic type to specific type at compile time
- errors will be at *compile time* instead of *runtime*

```java
class Pair<S, T> / class Pair<S extends Comparable<S>, T>
class DictEntry<T> extends Pair<String, T> {...}
public static <T extends String> T foo(T[] arr)
public <U> void printSomething(U value)
    @SuppressWarnings("unchecked")
    T[] a = (T[]) new Object[size];
    this.array = a;
A.<Circle>contains(...)
```

Notes

```java
B implements Comparable<B> {...}
A extends B {...}
A <: B <: Comparable<B> <: Comparable<?>
Comparable<A> INVARIANT Comparable<B>
Comparable<A> <: Comparable<? extends B>
Comparable<B> <: Comparable<? super A>
```

Type Erasure

During compilation, type parameters are erased and replaced with the most specific reference type
* `Object` for *unbounded* and the bound for *bounded* parameters
Heap pollution - situation where a parameterized type expects to hold an object of type X but stores an object of type Y instead
Suppress warnings - for type error and only applies to variable declarations
Reifiable types - full type info is available at run-time
* Generics are not reifiable due to type erasure

After type erasure, **ALL** wildcards and generic type params will be erased

Raw Types

`Seq<?>` - sequence of specific, but unknown types
`Seq` - sequence of Object instances, no type checking
`Seq<Object>` - sequence of Object instances

Wildcards

PECS - Producer *Extends*, Consumer *Super*

Upper-bounded `V<? extends A>` - to get values out of a structure

- Covariance
- if `S <: T`, then `A<? extends S> <: A<? extends T>`
- `A<S> <: A<? extends S>`

Lower-bounded `V<? super A>` - to put values into a structure

- Contravariance
- if `S <: T`, then `A<? super T> <: A<? super S>`
- `A<S> <: A<? super S>`

Unbounded `V<?>` - The parent class of all wildcards
* `Array<?>` is the supertype of all `Array<T>`

Type Inference

Ensures **type safety** - compiler can ensure that `List<myObj>` holds objects of type `myObj` at *compile time* instead of *runtime*

* `<? super Integer>` - inferred as `Object`
* `<? extends Integer>` - inferred as `Integer`

Constraints
1. Target typing - type of target variable
→ `Shape o = statement()`, so `T <: Shape`
2. Type parameter bound
→ `public static <T extends GetAreable>`, so `T <: GetAreable`
3. Argument parsed
→ `Seq<Circle> <: Seq<? extends T>`, so `T <: Circle`
→ `Shape <: S`, so `Shape <: S <: Object`

Most specific

`Type1 <: T <: Type2`	→ `T = Type1`
`Type1 <: T`	→ `T = Type1`
`T <: Type2`	→ `T = Type2`

Java Access Modifiers

- `public` - visible from outside the class
- `private` - only visible within enclosing class
- `protected (default)` - only visible within enclosing class and subclasses

- `static` - associate field with class
- `final` - value will not change (fields) / not inheritable (classes)

Immutability

Returns a new object every time something is modified
* requires explicit reassignment

Advantages

- **Ease of understanding** - obj not modified unless reassigned
- **Enabling safe sharing of objects**
- **Enabling safe sharing of internals**
  - `@SafeVarargs` - annotation to indicate varargs (`T... arg`) is safe
- **Enabling safe concurrent execution**

Nested Class

Static nested class - associated with the containing class, not instance
Inner class

- can access static & non-static parent fields & methods
- **qualified** `this` (`A.this.x`) - access `x` from `A` through `A.this` reference
- methods & fields of private classes not accessible outside enclosing class

Local class

- within method
- variable capture - variables used only within the scope (enclosed within `{}`) must be effectively final

Anonymous class

- `new Constructor(arguments) { body }`
- can extend only 1 class or interface, cannot extends & implements together
- **lambda** is a type of anonymous class, must be effectively final

Side Effect-free Programming

Pure functions - immutable and does only 1 thing, reproducible
`@FunctionalInterface` - interface with only 1 abstract method
Method reference - `Box::new` means `x -> new Box(x)`
Curried functions - args taken 1 by 1 (`f(a)(b)(c)` instead of `f(a,b,c)`)

Streams

Terminal operations (doesn't return a stream)
`reduce` accumulate items in stream - e.g. `reduce(0, (acc, val) -> acc + val)`
`forEach(Consumer<? super T> action)` / `count()` / `toList()`

Intermediate operations (returns a stream)
`filter()` / `flatMap()` / `map()` / `limit()` / `takeWhile()` /
`concat(Stream<? extends T> a, Stream<? extends T> b)` A+B stream
`peek(Consumer<? super T> action)` apply action to every element

Stateful / Bounded operations
`distinct()` / `sorted(Comparator<? super T> comparator)`

Element matching
`allMatch(Predicate<? super T> pred)` / `anyMatch()` / `noneMatch()`

Parallel
`parallel()` / `sequential()` / `unordered()`

Monad - supports `flatMap`

Left identity law - `Mondad.of(x).flatMap(x -> f(x)) == f(x)`
Right identity law - `monad.flatMap(x -> Monad.of(x)) == monad`
Associative law - order of grouping `flatMap` shouldn't matter

Functors - supports `map`

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 Streams

Concurrency - only one process running at a time, multiple processes making progress
Parallelism - multiple processes running at the same time
Ordered / Unordered source - `parallel()` on ordered stream is expensive

Parallelization criteria

1. **No interference** - source of stream must not be modified in terminal operation
2. **Stateless** - computation should not depend on previous execution results
3. **No side effects**
4. **Associative** - e.g. `reduce`
   - combiner and accumulator must be associative
   - combiner and accumulator must be compatible - `combiner.apply(u, accumulator.apply(identity, t)) == accumulator.apply(u, t)`

Asynchronous Programming

Threads

`new Thread(() -> body).start()` - initiates execution, returns immediately, doesn't wait for execution to complete
`Thread.currentThread().getName()` - return name of current thread
`Thread.sleep(x)` - sleeps for `x` milliseconds

CompletableFuture

`of() -> completedFuture(T value)`
`of() -> CompletableFuture<Void> runAsync(Runnable action)`
`of() -> CompletableFuture<T> supplyAsync(Supper<T> s)`
* for both `runAsync` and `supplyAsync` - completes when lambda finishes execution
`map() -> thenApply(Function<? super T, ? extends U> fn)`
`flatMap() -> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)`
`combine() -> thenCombine(CompletionStage<? extends U> other, BiFunction<? super T, ? super U, ? extends V> fn)`

`thenRun(Runnable r)` - Runnable executed after current stage completes
`runAfterBoth(CompletionStage<?> other, Runnable r)`
`runAfterEither(CompletionStage<?> other, Runnable r)`

`get()` - synchronous / `join()` - `get()` but doesn't throw checked exceptions

`handle((value, exception) -> (e == null) ? val : 0)`
* one of value or exception will be `null`

Notes

- `...Async()` only affects **how** the methods are called, sequence of operations **not affected**
- `...Async()` assigns task to new worker thread
- `then...Async()` run togther in parallel given enough worker threads
- CompletableFuture keeps a **stack** of lambda expressions to call next when completed
  - supplyAsync(lambda1); thenApply(lambda2); thenApply(lambda3) → **execution order:** lambda1, lambda3, lambda2

Fork & Join - `RecursiveTask<T>`

`fork()` - execute (`compute()`) item in another thread / adds task to front of deque (double-ended queue)
`join()` - blocks computation of thread until completed
Work stealing - idle threads steal from tail of other non-empty deques

Order

`left.fork()`, `right.fork()` ⇒ `right.join()`, `left.join()`

Stack & Heap

Heap

- one new object ⇒ one box
- anonymous classes like producer & transformer must write `anon`.
- lazy consist of producer, need create a new producer obj and link this.producer to it

Lambda - if in process of creating lambda, a stack frame is created for the method of interface you are assigning to (e.g. producer = () -> {inside here [produce stack frame is created]})

Misc

`ClassName::methodName` - method references
`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)`, depends on whether `A::foo` is an instance or class method

Maybe

1. `Maybe::map` can be used to traverse if `Maybe::Some`
2. `Maybe::orElseGet` can be used to traverse if `Maybe::None`
3. `Maybe::ifPresent` can be used to add element to data struct - `maybe.ifPresent(item -> struct.add(item))`
4. Adding multiple items in Maybe using `ifPresent` - `maybe1.ifPresent(item1 -> maybe2.ifPresent(item2 -> struct.add(item1, item2)))`

Streams

1. Use `Stream::map` to convert `Stream` into desired format - `intStream.map(i -> i.toString())`
2. Use `Stream::flatMap` to expand stream - `List.of([1, 2]).stream().flatMap(x -> Stream.iterate(x, y -> y * 2).limit(2))` returns `[1, 2, 2, 4]`