Abstract Classes

1. Cannot be instantiated
2. Can extend a concrete class
3. Concrete class must implement all abstract methods/declare them as abstract
4. Cannot be final

Interface

1. Fields are implicitly public & final
2. Methods are implicitly public & abstract

Misc.

1. Final fields, if not instantiated, need to be instantiated in the constructor
2. Static methods cannot use **“this”** or **“super”** keywords
3. Constructors cannot be final or final

OOP Principles

1. Abstraction (reduce complexity)
2. Encapsulation (protection)
3. Polymorphism
4. Inheritance

Tell-Don’t-Ask

class A should not ask for class B’s internals to perform computation. class A should *tell* class B to perform and return the result of the computation

Liskov Substitution Principle

Try to substitute all instances of A with subclass B and see if the result is still exactly the same. If it is, then it adheres to LSP.

CS2030S	java.util.function
`BooleanCondition<T>::test`	`Predicate<T>::test`
`Producer<T>::produce`	`Supplier<T>::get`
`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`

Method Overloading

1. Same name
2. Param differ in type/number/order
3. Class methods can be overloaded w instance/other class methods

Compile Time	Run Time
Type Inference Type Erasure Type Casting	Dynamic Binding
Type Checking Accessibility Check	

- Return type does not matter
- Will not compile if ambiguous

Method Overriding

Only for non-static methods

Same Method Signatures	Superclass' Instance Method	Superclass' Static Method
Subclass' Instance Method	Overrides	Compile-time error
Subclass' Static Method	Compile-time error	Hides

1. Same name
2. Exact same parameters
3. Same/subtype return type[1]
4. Cannot override a private with public and vice versa
5. supercls throws Checked Exception1: Can throw an Checked Exception2 <: Checked Exception1 or none or any Unchecked Exception
6. Cannot throw Checked Exceptions if supercls does not throws any

[1]int is not subtype of double
[2]private & private || private → public: co-exist, but different;
public & public: override;
public → private || package-protected: error

Dynamic Binding

CT:

1. Record compile-time type of target of invocation (T).
2. Record compile-time type of parameters (P).
3. Find in class T the method(s) that can accept paramter type P and are also visible from where the method is being called.
4. (Choose the most specific method if there are multiple)
5. Record the method descriptor

RT:

1. Determine run-time type S of target of invocation.
2. Find in class S the method with matching method descriptor from CT.5
3. If not found, climb up the inheritance tree

Method Signature: name, num params, param order and param type
Method Descriptor: Method Signature + return type

Type Casting

Occurs during compile time, but checked during run time.

Exceptions

Checked	Unchecked
Not within programmer's control	Within programmer's control
extends Exceptioon	extends RunTimeException
Must be handled	Need not be handled

1. If a method can throw a Checked Exception (either by calling another method that throws the Checked Exception or by throwing it explicitly), it must either:
   - Handle it by wrapping the statement that could throw the Exception in a try block and catching the Checked Exception
   - Contain the "throws" clause in method header that throws the Checked Exception or its supertype
2. If bar() invokes foo() which is declared to "throws" a Checked Exception, bar() must handle it as per above, regardless if it is already caught and handles by foo() itself (treat the JVM as blind)
3. Catch blocks must be arranged from most to least specific in order to compile properly
4. Catch blocks cannot catch Checked Exceptions that can never be thrown in the corresponding try block (although Exception is fine)

Control Flow

1. Any code below "throw" keyword will not be executed
2. If a catch block is executed, any code under try-catch-finally block will not be executed
3. A method can throw an Exception in its try block and catch it itself in its catch block. However, it cannot catch Exceptions it throws in its catch and finally blocks.
4. Finally block will run just before control is passed to another method (via return/throw). Therefore, return/throw statements in Finally block will override any return/throw statements in try/catch block.Z

Type Erasure

During **compile time** but after validation

- <T>: T becomes ctt Object
- <T extends Class> becomes ctt Class
- Stuff<T> becomes ctt Stuff

Narrow Type casting involving generic types will result in compile warning; T is type erased during CT, so the runtime system cannot safely check the type to make sure that is matches and trusts that the programmer is correct
When a generic method is called (eg T get()), java automatically type casts the returned object to T.
Improper casting can lead to ClassCastException [Integer i = (Integer) "foo"] or Incompatible type error [String s = (Integer) "foo"].

ArrayStoreException

```
Integer[] intArray = new Integer[10];
Object[] objArray = intArray;
objArray[0] = "foo";
```

Heap Pollution – no ArrayStoreException (explains why Java doesn't allow array of generic classes/types)

```
Pair<Integer, String> pairArray = new Pair<>[];
Object[] objArray = intArray;
objArray[0] = new Pair<Double, Boolean>(1.1, true); // no excpetion due to type erasure
```

Variance

Covariance: S <: T → C(S) <: C(T) (e.g. Arrays)
Contravariancee: S <: T → C(T) <: C(S)
Invariant: S <: T → No relationship between C(S) and C(T)

Wildcards

Objects cannot be instantiated with wildcards
i.e. new A<?>() → CT Exception
Subtyping
T <: S
A<T> <: A<? extends T> <: A<? extends S>
A<T> <: A<? super T>

ArrayList<T> <: List<T> <: List<? extends T> <: List<? extends S>
<: List<? super T>

ArrayList

ArrayList<T>: can add anything <: T; can reference as >: T
ArrayList<? extends T>: can only add null; can reference as >: T
ArrayList<?>: can only add null; can only reference as Object
ArrayList<? super T>: can add anything <: T; can only reference as Object

Arrays

new T[10]; new A<T>[10]; error
new A<?>[] → the only acceptable generic array; can add anything <: A
A<?> same as A<? extends Object>

Nested Wildcards

ArrayList<A<Integer>> </: ArrayList<A<Number>>
ArrayList<A<Integer>> </: ArrayList<? extends A<Number>>
ArrayList<A<Integer>> <: ArrayList<? extends A<? extends Number>>

Warnings

A rawA = new A(); **A<T> paramA = rawA;** Assigning a parameterized type to a raw type. Value "stored" in rawA might not be type T, but calling "T get()" on paramA has implicit type casting to (T) get(). This can result in	**A a = new A<T>();** No warning upon instantiation. But may result in "incompatible types" runtime error because generic methods return type "Object" (no implicit type cast)	**...** **T t = (T) someObject;** Unchecked warning because T is type erased during CT, so runtime system cannot safely check if someObject is of type T

Pure Functions

Side-Effect free:
1. Deterministic
2. No print statements
3. Cannot modify non-local params
4. Cannot throw Exceptions
   - Math.sqrt(-1): returns Nan
   - (double/float) x / 0: returns Infinity

Inner Class

Only nest it if it is only used within the Outer class
1. No static fields except constants
2. Can access ALL Outer class's fields (incl. private)
   - can't use "this", but can use "Outer.this"
3. If a field in Inner class has same name as one from Outer class, the one in Inner one takes precedence
4. Outer class can acess ALL Inner class's fields (incl. private)
5. Instantiation (subject to visibility):
   Diff. package: Outer.Inner inner = new Outer().new Inner();
   Static method in Outer: Inner inner = new Outer().new Inner();
   Instance method in Outer: Inner inner = new Inner();

Nested (static) Class

1. Can declare static fields
2. Can access ALL Outer class's STATIC fields
3. Outer class can acess ALL Inner class's fields (incl. private)
4. Instantiation (subject to visibility):
   Diff. package: Outer.Inner inner = new Inner();
   Same package: Inner inner = new Inner();

Local Class

1. No fields except constants
2. Can access final/effectively final fields local to the enclosing method. Local classes also cannot change the reference to these fields (but can eg change A[0])
3. Static enclosing method: Can access all Outer class's static fields
4. Non-static enclosing method: Can access all Outer class's fields
5. Cannot have access modifiers and cannot be static
6. **Variable Capture:** creates a copy of final/effectively final local variables (variables in the enclosing method), as well as a reference to the outer class

Anonymous Class

1. Same as Local Class, but must **extend** something
   eg. new Comparator<String>() {...};

Lambdas

1. An anonymous class that implements a Functional Interface. The body will be the implementation of the interface method
2. **Closure:** Final/effectively final variables that are used in computation are captured by the lambda expression. As such, lambdas can only use final/effectively final local variables

Method Referencing

1. Static methods of a class:
   A::foo == (x, y) -> foo.(x, y)
   == (x, y) -> x.foo(y) (instance method)
2. Instance methods of a class/interface:
   a::foo == y -> x.foo(y)
3. Constructor of a class:
   A::new == x -> new A(x);

Stack & Heap

- **Stack:** each stackframe = each function call, including constructors; Each frame contains local vars, this & params. For main, remember String[] args
- **Heap:** each instance object = each box; instance fields inside
  - For instances of inner class: always captures qualified "this" ref.
  - For instance of local class: always captures qualified "this"; captures local vars if needed
- **Metaspace:** stores static attributes (excl. nested classes); don't need a box for the class

Take note of the number of instances on the heap (mutable vs immutable)!

Currying

A sequence of unary functions composed together eg. x -> y -> z -> x + y + z;
Used for *partial* evaluation of *f(x, y, z)*, where y and z are computationally expensive.

Stream Operations

Construction	Intermediate	Terminal (used once)
Stream.of()	map(Function<? super T, ? extends U>)	forEach()
Stream.generate(Suppier<T>)	filter(Predicate<? super T>)	reduce(T identity, BinaryOperator<T>)
Stream.iterate(T seed, UnaryOp<T>)	flatMap(UnaryOp<T>)	noneMatch(Predicate<T>)
Arrays.stream()	limit(int)	allMatch(Predicate<T>)
List.stream()	takeWhile(Predicate<T>)	anyMatch(Predicate<T>)
IntStream.range(int i, int j)	peek(Consumer<T>)	

Monads (of + flatMap)

1. **Left Identity**: Monad.of(x).flatMap(x -> f(x)) == f(x)
2. **Right Identity**: monad.flatMap(x -> Monad.of(x)) == monad
3. **Associative**: monad.flatMap(x -> f(x)).flatMap(x -> g(x)) == monad.flatMap(x -> f(x).flatMap(g(x)))

Functors (map)

1. **Identity**: functor.map(x -> x) == functor
2. **Composition**: functor.map(x -> f(x)).map(x -> g(x)) == functor.map(x -> g(f(x))

Parallel Streams

Embarassingly Parallel: Each element in the stream is processed individually w/o depending on other elements; little to no communication between parallel tasks

1. **No Interference** with Stream elements
2. **Stateless**: should not depend on the state of any stream element
3. **No Side-effects**: should not modify anything outside (eg. add stuff to ArrayList which could lead to Race Conditions)

Reduce

Accumulator: Combines partial result & next element in stream
Combiner: Combines 2 partial results
Rules:
1. combiner.apply(identity, i) == i
2. combiner and accumulator must be associative i.e. f(f(a, b), c) == f(a, f(b, c))
3. combiner and accumulator must be compatible: combiner.apply(u, accumulator.apply(identity, t) == accumulator.apply(u, t)

Threads

- One flow of execution per thread;
- Does not do anything until .start(), which returns immediately
- Drawbacks: Overhead, Handling Exceptions, Passing computations between Threads

CompletableFuture

- allOf(): returns a CompletableFuture<Void> if all CFs are completed
  - .thenRun(): runs the Runnable when all CFs completed
  - .join(): acts as a blocker to ensure all CFs are completed before the code below can run

- join()
  - join() blocks and waits for the target CF to be completed.

```
static CompletableFuture<Void> printAsync(int i) {
  return CompletableFuture.runAsync(() -> {
      doSomething();
      System.out.print(i);
  });
}

public static void main(String[] args) {
  printAsync(1);
  CompletableFuture.anyOf(printAsync(2), printAsync(3))
    .thenRun(() -> printAsync(4))
    .join();
  doSomething();
}
```

```
static CompletableFuture<Void> printAsync(int i) {
  return CompletableFuture.runAsync(() -> {
      doSomething();
      System.out.print(i);
  });
}

public static void main(String[] args) {
  printAsync(1);
  CompletableFuture.allOf(printAsync(2), printAsync(3)).join();
}
```

LEFT: completed when CompletableFuture.anyOf.. is completed, which is when printAsync(4) returns. The returned CF from printAsync(4) may or may not be completed.

RIGHT: completed when CompletableFuture.allOf(...) is completed, which is when both printAsync(2) & printAsync(3) are COMPLETED (not just returned)