☐☐ **Abstraction Principles** ★

We can assess abstractions by:

- **Expressive Power**: What range of things you can do?
- **Complexity**: How complicated is it to understand/use?
- **Obscurity**: Is there hidden knowledge you might need?
- **Depth**: How much complexity does it hide?

☐☐ **Abstraction Tradeoffs** ★

- Expressive Power vs. Complexity
- Obscurity vs. Complexity/Depth
- Expressive Power vs. Depth

Lec 5 | Apply to recent material.

☐☐ **Subtype Polymorphism**

May use varied implementations through a single base interface.

The set of subtypes is **opaque**:

- To **use** subtypes, we only depend on the base interface.
- Subtypes must follow the **Liskov Substitution Principle (LSP)**.

Lec 9 | Lab 7

☐☐ **Metaprogramming** ★

A program treats itself as data via:

- **Introspection**: Inspect/Reason.
- **Intercession**: Modify behavior.
- **Generation**: Create new code.

Approach varies significantly between languages. Each part may occur at *compile-time* or *runtime*.

Examples of metaprogramming:

- Generate parts of code that share a type, e.g. `Menu<Result>` contains `MenuItem<Result>` and has a `prompt()` function that returns `Promise<Result>`.
- Write a "`getSize()`" function that inspects a type for `.size()`, `.length()`, or neither, then the code dynamically (intercedes) to access the right one or throw.
- For `Observable<ObserverT>`, generate code with a custom, type-safe `emit()` function for each handler in `ObserverT`.

Lec 12 | Identify and evaluate uses.

☐☐ **What is Coupling?** ★

- Dependencies (one- or two-way)
- Require changing together
- Reliance on shared knowledge, impl. details, or design decisions

☐☐ **Kinds of Coupling** ★

- **Mechanical**: Shared environment
- **Structural**: Availability of data and operations (types/names)
- **Behavioral**: Requirements, results, side effects, or timing
- **Semantic**: Meaning of data or operations, or interpretation

☐☐ **Impact of Coupling** ★

- **Complexity**: How complicated?
- **Distance**: How far separated?
- **Feedback**: Reliably detect issues?
- **Volatility**: How likely to change?

Lec 6 | Apply to recent material.

☐☐ **Ad Hoc Polymorphism**

Functions may operate on any variants belonging to a union type.

The set of variants is **known**.

- We consider each variant one-by-one when **using** them.

Lec 9 | Lab 7

☐☐ **TS Metaprogramming** ★

Often combines *compile-time type introspection+generation* with *runtime introspection+intercession*.

- **Type Manipulation**: use mapped types, conditional types, type guards, `keyof T`, indexed access (`T[K]`), `infer`, etc. to generate compile-time types for the intended interface and behavior.
- **Runtime Implementation**: use `typeof`, `in` operator, property checks, branches/loops, etc. to implement desired runtime behavior. May (cautiously) use casts internally where safety is already verified at the type-level.

Use **decorators** to modify/generate code attached to parts of classes.

Lec 12 | Lab 10 | Review ideas and patterns. We'll remind you of TS specifics where needed on the exam.

☐☐ **Encapsulation** ★

For each unit of encapsulation:

- Dependencies freely allowed inside the unit.
- Code outside interacts only through interface.

Lec 7 | Apply to recent material.

☐☐ **Parametric Polymorphism**

Use of templates/generics.

- Variation via type parameters, e.g. `Grid<T>`
- Constraints, e.g. `<Kind extends PuzzleKind>`

Lec 9, 12

☐☐ **Subtype vs. Ad Hoc vs. Parametric** ★

Parametric polymorphism can be used on its own, or combined with either approach. Consider:

- Use only when subtype or ad hoc alone is insufficient. (See metaprogramming below.)
- Use on its own (no constraints) if the generic class/function is just a "carrier", like `Grid<T>`.
- Constrain generic types with a base interface or union type if you need specific functionality.

Lec 12 | Identify appropriate places to apply each.

☐☐ **C++ Metaprogramming** ★

Templates enable *compile-time* code *generation*.
e.g. **SFINAE**: *Substitution Failure Is Not An Error*:

- **Overload Cases**: templated overload with code for each case we need to support.
- **Match/Filter**: the signature of each only compiles for its target types, effectively generating based on conditional introspection.

Lec 12 | Review ideas and patterns. We'll remind you of C++ specifics where needed on the exam.

☐☐ **Concurrency** ★★

A program may manage several tasks together to:

- **Improve Responsiveness**: Prevent long tasks from blocking time-critical or user-facing tasks.
- **Mitigate Latency**: Work on other tasks while waiting for I/O or other external resources.
- **Increase Throughput**: Get more done with existing or additional computational resources.

Challenges:

- Increased complexity and behavioral coupling.
- Additional pitfalls, such as **race conditions** (there are others, but we didn't cover them).
- Complex interaction with error handling.

Lec 16, 17, 18 | Lab 12 | Be prepared to evaluate application of concurrency to different scenarios, assess trade-offs, and diagnose potential issues.

☐☐ **Inversion of Control** ★

Definition: Direction of control flow opposite the order of dependency.

Common Design Patterns:

- **Observer:** Observers are directly registered with a subject and receive callbacks for events.
- **Publish/Subscribe:** Subjects publish events to a broker that broadcasts to subscribers. Events often organized by topics.

Lec 8 | Lab 11 | Project 4

☐☐ **Error Handling Challenges** ★

Error cases -> *partial* abstractions:

- May be **obscure** if the error case isn't obvious or is easy to forget.
- May be more **complex** for client code to address the error case.

Partial abstractions affect *coupling*:

- All dependents are also partial, unless somehow insulated.
- Encapsulation boundaries can complicate error handling.

Lec 13 | Be prepared assess impacts.

☐☐ **Error Handling & Design** ★★

Taxonomy of Exceptionality:

- **Fatal**: unrecoverable system error (e.g. out of memory).
- **Preventable**: programming error that could/should be fixed.
- **Exogenous**: operation/resource out of our control fails.
- **Vexing**: thrown due to poor design, wrong encapsulation, etc.

Catch and handle exogenous errors.

If you notice a preventable error, don't catch it - just fix the bug!

Design/refactor to remove vexing exceptions (where you can).

Philosophies:

- **LBYL**: *Look Before You Leap*. Check preconditions first.
- **EAFP**: *Easier to Ask for Forgiveness than Permission*. Attempt the operation and handle errors that come up.

Prefer LBYL for preventable errors and EAFP for exogenous errors.

Lec 13, 14 | Lab 9 | Project 4 | Analyze and evaluate examples.

☐☐ **Approaches to Errors** ★★

Exceptions

`throw` exceptions to signal errors.

- "Happy Path" is clearly encoded.
- Error paths are **implicit**: jump to a `catch` or propagate outward.
- Some languages support **typed** or **checked** exceptions.

Errors as Values

`return` errors as a distinct values

- All control flow are **explicit**.
- Interleaved "Happy" + "Error" paths. Manual propagation only.
- Some languages use a checked **result type**, e.g. `Result<T,E>`.

Lec 13, 14 | Be fluent in both styles and their strengths or weaknesses.

☐☐ **Error/Exception Safety** ★★

A function can't *complete normally* when it can't compute its return data or ensure its postconditions.

Possible error safety guarantees:

1. **No-Fail**: Always works normally.
2. **Strong**: If an error thrown / returned, no state is modified.
3. **Basic**: If an error thrown / returned, no invariants broken.
4. **None**: If an error thrown / returned, all bets are off. May even leak resources.

Prepare/Commit Pattern

- First, attempt operations that might fail, often on a temp copy.
- Only if successful, apply changes via e.g. a swap that can't fail.
- Example: Copy-Swap in C++.

Lec 14 | Be prepared to analyze code and which guarantee it provides.

☐☐ **Resource Management** ★★

A **resource** is anything that must be *acquired*, used, and *released*.

- open files
- locks
- pizza robots
- dyn. memory
- observer subs
- connections

Avoid **leaks** or **dangling** resources:

- **`try/finally`**: manual cleanup
- **Structured Resource Manager**: e.g. C++ destructors (RAII) e.g. TS `using` + `Symbol.dispose`.

Lec 15 | Identify resources and issues.

☐☐ **Concurrency Terms and Concepts** ★

Concurrent: Interleaved/overlapping tasks.
Parallel: Multiple tasks executing at once.

Thread: A sequence of program execution.
Single-threaded: One thread. One task at a time.
Multithreaded: Multiple threads allow parallelism.

A single-thread may switch between tasks rapidly or interface with external, parallel resources.

Multitasking: Switching between multiple tasks...
Cooperative: Tasks must yield control voluntarily.
Preemptive: Scheduler decides and can interrupt.

Communication between tasks can be done with:
Shared Memory: Read/write shared variables.
Message Passing: Send messages, copying data.

Lec 16, 17 | Identify these terms/concepts in various programs. Assess tradeoffs at a high level.

☐☐ **Synchronization** ★★

Synchronous: A -> B guaranteed ordering.
Asynchronous: Indeterminate ordering.

Programmers must ensure correctness without overly constraining asynchronous efficiency gains.

Blocking: Control flow pauses to wait for I/O.
Non-Blocking: Request I/O, control flow moves on. Check back later or register a callback.

Blocking or not influences synchronization.

Lec 16, 17 | Lab 12 | Analyze synchronization in code and identify potential improvements or issues.

☐☐ **Cooperative Multitasking** ★★

Multiple tasks coordinate and yield appropriately.

- **Generators / Coroutines**: `yield`, resume later
- **Callbacks**: Register continuation callback. Awkward nesting and separate error callbacks.
- **Promises**: Object for future results/errors, specify continuation via `.then()/.catch()`.
- **`async/await`**: `await` waits for a promise to resolve, then rest of function is a continuation.

Lec 17 | Pizza Restaurant Examples | Be familiar with each, no need to memorize specific syntax or details.

☐☐ **Structured Concurrency** ★

Restrict concurrency to well-defined, recognizable patterns - analogous to structured programming:

- A task is owned by a scope or "nursery" object.
- No tasks outlive owner. No background tasks.
- All "split" concurrent tasks come back together.
- Owner waits on all tasks before cleanup.
- Automatic error propagation and cancelation.

Lec 18 | Understand motivations for each and potential problems with unstructured concurrency.