C#/.NET Interview Cheat Sheet - Senior Research Software Engineer

Static Class

- Cannot be instantiated. Used for utility or helper functions.
- Members must be static. No instance members allowed.
- Real-world example: A `CloudLogger` class that logs messages to AWS CloudWatch.

```
public static class CloudLogger {
    public static void Log(string msg) => Console.WriteLine(msg);
}
```

Constructors (Default, Parameterized, Static, Copy, Private)

- Default: No parameters. Called implicitly.
- Parameterized: Accepts args. Used to inject dependencies (e.g., DbContext).
- Static: Initializes static members. Called once.
- Copy: Creates a new object using an existing one.
- Private: Prevents instantiation. Used in Singleton.

Example:

```
public class DbConnection {
    private DbConnection() {}
    public static DbConnection Instance { get; } = new DbConnection();
}
```

Class vs Record vs Struct

- Class: Reference type. Inherits. Mutable. Used for entities in web CRUD apps (e.g., User, Product).
- Record: Reference type (C# 9+). Immutable by default. Value equality. Good for DTOs.
- Struct: Value type. Immutable. No inheritance. Efficient for small data.
- Choose Class for business logic, Record for data transfer, Struct for lightweight configs.
- Avoid Struct for large data (copying overhead), and Record when mutability is needed.

Pros and Cons

- Class: Pros - OOP support, flexible. Cons - GC overhead.
- Record: Pros - Immutable, thread-safe. Cons - Limited mutability.
- Struct: Pros - No heap allocation, fast. Cons - No inheritance, can lead to boxing.

Exception Handling Best Practices

- Catch only specific exceptions. Avoid catching base `Exception`.
- Always log context (e.g., userId, requestId).
- Use finally or 'using' for cleanup.
- Real-world (CRUD + cloud): Wrap data save logic with try/catch to log errors to cloud:

```
try {
    dbContext.SaveChanges();
} catch (DbUpdateException ex) {
    CloudLogger.Log(ex.ToString());
    throw;
}
```

throw vs throw ex

- `throw;` preserves original stack trace - preferred for rethrowing.

- `throw ex;` resets stack trace - avoid unless modifying exception.
- Example:

```
try {
    UpdateUser(user);
} catch (Exception ex) {
    Log(ex);
    throw; // use throw to preserve error origin
}
```

- Use `throw ex;` only when you must enrich or wrap the exception with new context.

throw vs throw ex (Deep Dive)

- `throw;` is used to rethrow the original exception while preserving the original stack trace.
- `throw ex;` creates a new stack trace, hiding where the original exception occurred-bad for debugging.

Use `throw ex;` only when you **enrich or wrap** the original exception with additional context.

Example - Wrapping with context:

```
try {
    var user = cloudClient.FetchUser(id);
} catch (HttpRequestException ex) {
    throw new Exception($"Failed to fetch user with ID {id}", ex);
}
```

- This helps downstream code or logs show both the error and the context in which it failed.

Avoid this:

```
try {
    var user = cloudClient.FetchUser(id);
} catch (HttpRequestException ex) {
    throw ex; // Resets stack trace - harder to debug!
}
```

Preferred if you're not modifying the exception:

```
try {
    var user = cloudClient.FetchUser(id);
} catch (HttpRequestException ex) {
    Log(ex);
    throw; // Keeps original stack trace intact
}
```