Part01

Q1: What are the benefits of using a generic sorting algorithm over a non-generic one?

✅ Benefits:

1. **Reusability** – same code works for any type (int, string, Employee ...).
2. **No code duplication** – you don't need separate methods per type.
3. **Performance** – avoids boxing/unboxing with value types.
4. **Type safety** – errors are caught at compile time.

Q2: How do lambda expressions improve the readability and flexibility of sorting methods?

✅ They:

- Make code **shorter and cleaner** (inline instead of external classes).
- Allow **quick customization** of sorting logic.
- Keep the comparison logic **close to where it's used**, improving clarity.

Q3: Why is it important to use a dynamic comparer function when sorting objects of various data types?

✅ Because:

- Different types require **different sorting rules** (salary, length, etc.).
- The comparer makes sorting **flexible and customizable**.
- It avoids modifying classes just to add sorting logic.

Q4: How does implementing IComparable<T> in derived classes enable custom sorting?

✅ Because:

- Each object knows how to **compare itself** to another.
- Sorting methods (Array.Sort, List.Sort) can directly use CompareTo.
- Provides a **natural ordering** for each class.

Q5: What is the advantage of using built-in delegates like Func<T, T, TResult> in generic programming?

✅ Advantages:

- **Saves time** – no need to define custom delegates.
- **Flexible** – works with any type.
- **Integrates with LINQ** seamlessly.
- **Clear naming** – easier to understand and maintain.

Q6: How does the usage of anonymous functions differ from lambda expressions in terms of readability and efficiency?

✅ Differences:

- **Anonymous functions** use the delegate keyword → more verbose.
- **Lambda expressions** are **shorter and more modern**.
- **Performance** is nearly the same, but lambdas are more readable.

Q7: Why is the use of generic methods beneficial when creating utility functions like Swap?

✅ Because:

- Works with **any type** without duplication.
- Ensures **type safety** (compile-time checking).
- Avoids writing separate methods for int, string, etc.

Q8: What are the challenges and benefits of implementing multi-criteria sorting logic in generic methods?

✅ Benefits:

- Provides **fairer and more accurate sorting** (e.g., Salary then Name).
- Flexible for **complex objects**.

✅ Challenges:

- Comparison logic can become **more complex**.
- **Performance concerns** with large datasets.

Q9: Why is the default(T) keyword crucial in generic programming, and how does it handle value and reference types differently?

✅ Because:

- Provides the **default value** of any type without knowing it at compile time.
- **Value types** → default is 0 / false / struct default.
- **Reference types** → default is null.

Q10: How do constraints in generic programming ensure type safety and improve the reliability of generic methods?

✅ Because:

- Constraints like where T : ICloneable ensure **required methods exist**.
- Prevent **runtime errors** by enforcing rules at compile time.
- Make generic methods **more reliable and predictable**.

Q11: What are the benefits of using delegates for string transformations in a functional programming style?

✅ Benefits:

- Transformation logic can be **passed as parameters**.
- Reduces code duplication (one method works for uppercase, reverse, etc.).
- Improves **modularity and reusability**.

Q12: How does the use of delegates promote code reusability and flexibility in implementing mathematical operations?

✅ Because:

- One function can handle **any operation** (add, subtract, multiply ...).
- Easy to add **new operations** without modifying core logic.
- Encourages **plug-and-play behavior**.

Q13: What are the advantages of using generic delegates in transforming data structures?

✅ Advantages:

- Single transformation logic works for **any type**.
- Reduces code duplication.
- Makes it easy to **convert or reshape data** dynamically.

Q14: How does Func simplify the creation and usage of delegates in C#?

✅ Because:

- Provides a **standard delegate type** for functions with return values.
- No need to define custom delegate types.
- Works seamlessly with **LINQ and lambda expressions**.

Q15: Why is Action preferred for operations that do not return values?

✅ Because:

- Action<T> clearly indicates a **void-returning operation**.
- Improves readability – makes the purpose of the delegate obvious.

Q16: What role do predicates play in functional programming, and how do they enhance code clarity?

✅ Because:

- A **Predicate<T>** represents a condition (bool result).
- Makes filtering and searching **clear and expressive** (Find, Where).
- Improves readability of code that deals with conditions.

Q17: How do anonymous functions improve code modularity and customization?

✅ Because:

- Logic can be **defined inline** without creating extra methods.
- Enables **custom, one-time use logic**.
- Keeps code modular and easy to adapt.

Q18: When should you prefer anonymous functions over named methods in implementing mathematical operations?

✅ Use anonymous functions when:

- The operation is **simple and short**.
- It's **only used once** and doesn't need reuse.
- You want to **keep the code compact**.

Q19: What makes lambda expressions an essential feature in modern C# programming?

✅ Because:

- They're **concise and expressive**.
- Power features like **LINQ, async, events, delegates**.
- Enable a **functional programming style** in C#.

Q20: How do lambda expressions enhance the expressiveness of mathematical computations in C#?

✅ Because:

- Complex operations can be written in **one short line**.
- Easy to pass as parameters to methods.
- Provide **runtime flexibility** for defining operations.

Part02+Bonus

1. Parallel Programming & Concurrency

Key Concepts

- **Concurrency** enables multiple tasks to make progress simultaneously (via time-slicing or task switching), enhancing responsiveness and throughput getsdeready.comWikipedia.
- **Parallelism** involves executing tasks truly simultaneously across multiple processors or cores—ideal for compute-heavy workloads getsdeready.comblog.seancoughlin.me.

Why They Matter

- Concurrency improves **responsiveness**, especially for I/O-bound tasks and user-facing systems.
- Parallelism boosts **performance** for CPU-intensive tasks by leveraging multi-core hardware getsdeready.comblog.seancoughlin.me.

Benefits & Challenges

- **Benefits**: Multithreading enables responsive UI and better resource utilization AlgoCademyLearn Coding USAWikipedia.
- **Challenges**: Concurrent systems introduce complexity like race conditions, deadlocks, synchronization needs AlgoCademyLearn Coding USA.
- **OOP & Concurrency**: Integrating object-oriented design with concurrency constructs helps build scalable and maintainable systems Science & Tech Powered by AI.

---

2. Unit Testing & Test-Driven Development (TDD)

Definitions

- **Unit Testing** involves testing smallest pieces of code (e.g., methods/functions) in isolation to verify correctness Wikipedia.
- **Test-Driven Development (TDD)** is a methodology where you:
    1. Write a failing test (Red),
    2. Implement just enough code to make it pass (Green),
    3. Refactor the code (Refactor),
    4. Repeat GeeksforGeeksWikipedia.

Benefits

- **Unit Testing** catches bugs early, enforces better design, reduces cost of fixing defects, and supports frequent releases Wikipedia.
- **TDD** ensures that functionality works as intended from the start, drives cleaner design, and can lower defect rates SpringerLinkWikipediaArbisoft.

Clarification

- While unit testing verifies code after writing, TDD prescribes writing tests *before* writing the corresponding feature, ensuring tight integration between tests and implementation blog.machinet.netWikipedia.

---

3. Asynchronous Programming with async and await in C#

What It Is

- C# uses async and await to simplify asynchronous code, reducing blocking and improving responsiveness, especially in I/O-bound programs Microsoft LearnC# CornerZero To Mastery.

- Asynchronous operations are represented by Task or Task<T>, and await pauses execution until the task completes without blocking the thread C# CornerNDepend BlogWikipedia.

How It Works Under the Hood

- The compiler transforms methods using async/await into internal **state machines**, enabling non-blocking execution while preserving a readable, sequential coding style Microsoft for DevelopersNDepend Blog.

Why It Matters

- Enhances **UI responsiveness** and scalability by keeping the application thread free for other tasks SitePointNDepend Blog.
- Key for modern app development—especially in web, desktop, and mobile apps—to deliver smooth user experiences Zero To MasterySitePoint.
- Must be used carefully to avoid issues like deadlocks (e.g., due to misuse of .Wait()/.Result) and to handle exceptions correctly SitePoint.

Summary Table

Topic	Core Idea	Benefits	Considerations / Challenges
Concurrency & Parallelism	Managing vs. truly running tasks simultaneously	Responsiveness, throughput, scalability	Synchronization, complexity, debugging
Unit Testing & TDD	Validating units / Driving design via tests	Prevents bugs, enforces design, reliability	Discipline needed, initial overhead
Async / await (C#)	Non-blocking I/O and task handling	Smooth UIs, resource efficiency	Deadlocks, error handling, debugging

what is Asynchronous programming

Asynchronous programming is a **programming paradigm** that allows tasks to run **independently of the main program flow**, enabling other operations to continue without waiting for those tasks to finish.

Instead of blocking the program until a task (like reading a file, making a web request, or querying a database) completes, asynchronous programming lets the program handle other work and then return to the task once it's done.

🔑 **Key Points:**

1. **Non-blocking execution**
    - The program doesn't pause while waiting for long-running tasks.
2. **Callbacks, Promises, async/await**
    - These are common mechanisms used to handle asynchronous code in different languages.
    - In **C#**, async and await make asynchronous code look like normal sequential code.
3. **Better performance**
    - Especially useful in applications with many I/O operations (file system, network calls, database).
    - Prevents the UI from freezing in desktop/mobile apps.

✅ **Example in C#:**

```
public async Task GetDataAsync()
{
  Console.WriteLine("Fetching data...");
  string data = await File.ReadAllTextAsync("file.txt"); // Non-blocking
  Console.WriteLine($"Data: {data}");
}
```

🔷 Here, await lets the program continue running while File.ReadAllTextAsync reads the file in the background.

📌 **Benefits:**

- Improves **responsiveness** of applications.
- Efficient use of resources (CPU is not idle while waiting).
- Scales better in apps handling many requests (like web servers).