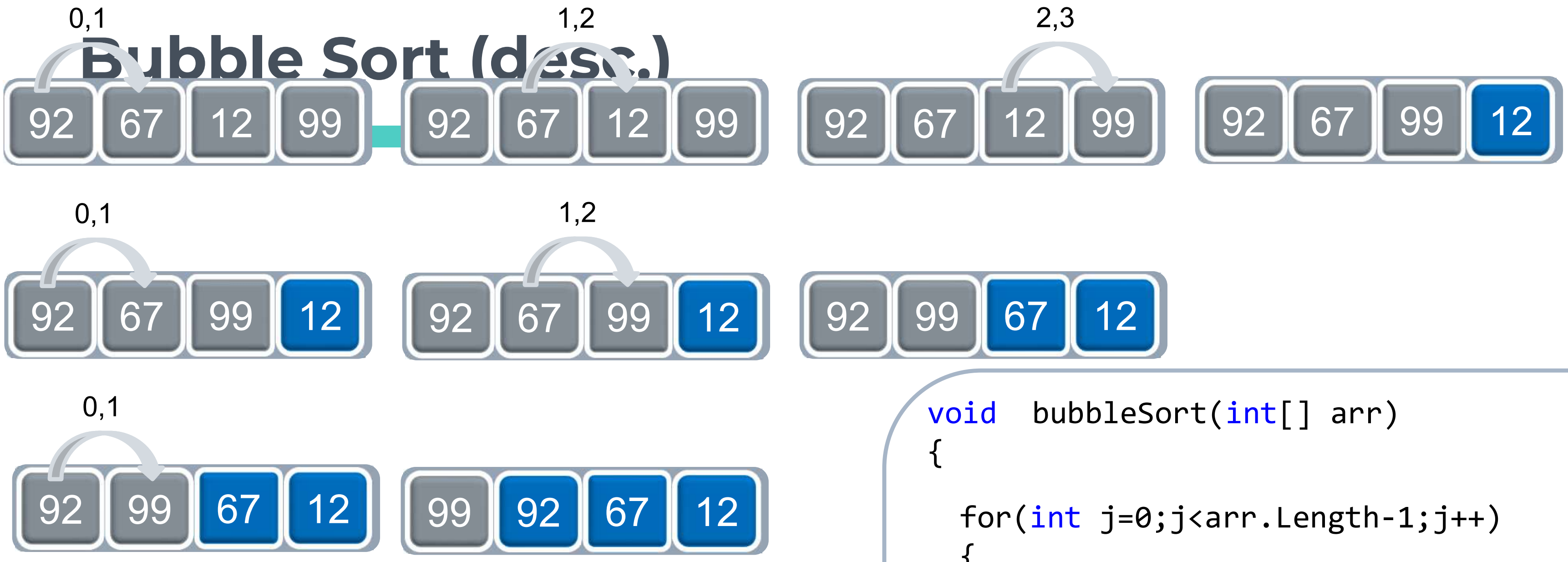
Day 6

Delegates and Events

Agenda for Day 6

- What are Delegates?
- Declaring, Instantiating, and Invoking Delegates
- Multicast Delegates
- What are Events?
- Event-Driven Programming Model
- Declaring, Raising, and Handling Events
- Built-in Delegates (**`Action`**, **`Func`** , **`Predicate`**)

Bubble Sort (desc.)

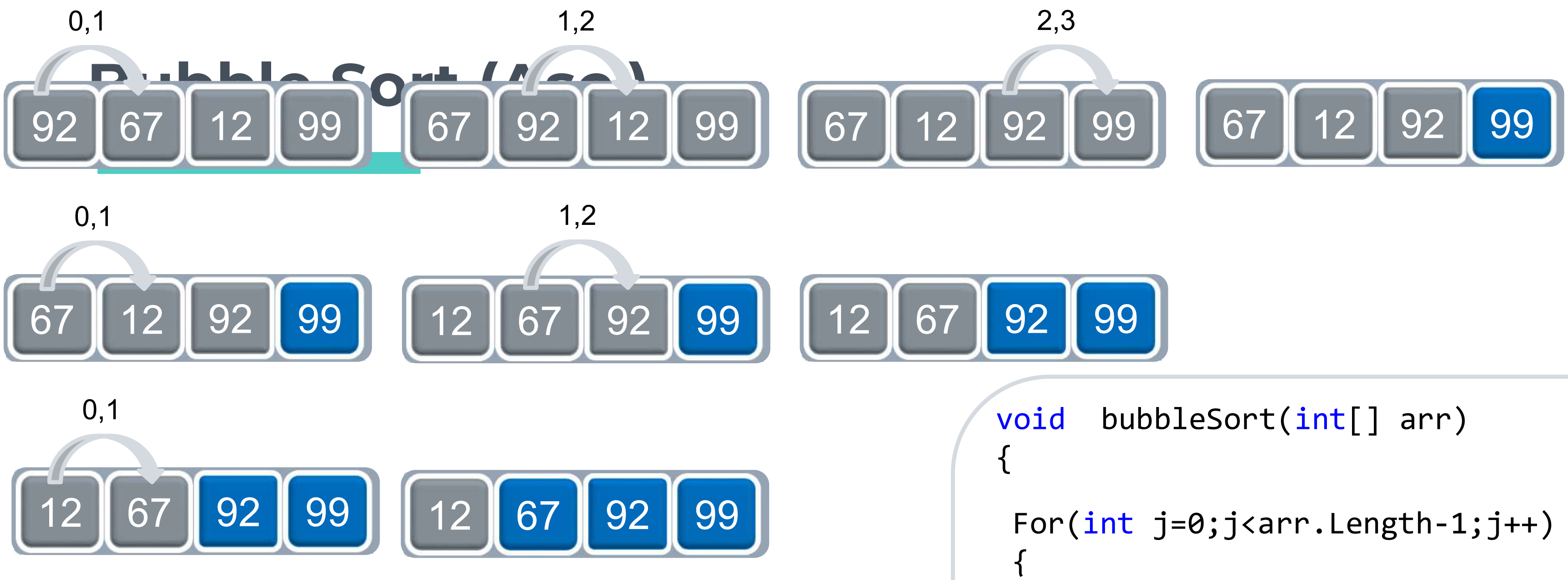


```
void  bubbleSort(int[] arr)
{

  for(int j=0;j<arr.Length-1;j++)
  {
    for(int i=0;i<arr.Length-1 -j  ;i++)
    {
      if(arr[i]<arr[i + 1])
      {
       swap(ref arr[i],ref arr[i+1]);
      }
    }

  }
}
```

Bubble Sort (Asc)



```
void  bubbleSort(int[] arr)
{

 For(int j=0;j<arr.Length-1;j++)
 {
  For(int i=0;i<arr.Length-1  -j  ;i++)
  {
    if(arr[i]>arr[i + 1])
    {
      swap(ref arr[i],ref arr[i+1]);
    }
  }
 }

}
```

Why Delegate

```
static bool sortAscending(int L, int M)
{
        return (L > M);
}
```

```
static bool sortDescending(int L, int M)
{
        return (L < M);
}
```

```
void  bubbleSort(int[] arr,          )
{
  for(int j=0;j<arr.Length;j++)
  {
    for(int i=0;i< arr.Length-1-j;i++)
     {
       if (                     )
       {
           Swap(ref arr[i],ref arr[i+1]);

       }
     }
  }
}
```

What is a Delegate?

- **Definition**: A type that represents references to methods with a specific signature.
- "Type-safe function pointers" or "method pointers.“
- Enables passing methods as arguments.
- Foundation for event handling.

Delegate Declaration

- **Syntax**:

```
public delegate [return_type] DelegateName([parameters]);
```

- Defines the signature (return type and parameters) that methods must match.
- **Example**:

```
public delegate void NotificationDelegate(string message);
```

Delegate Instantiation and Invocation

- **Instantiation**:
- `NotificationDelegate del = new NotificationDelegate(MyMethod);`
- Shorthand: `NotificationDelegate del = MyMethod;`
- **Invocation**:
  - Call the delegate instance like a method: `del("Hello!");`
- **Methods that can be assigned**: Static, Instance, Lambda Expressions.

What are Events?

- **Concept**: A mechanism for communication between objects.
- **Publisher**: The object that raises (publishes) the event.
- **Subscriber**: The object that listens for (subscribes to) the event and reacts.
- **Event-Driven Programming**: Program flow determined by events.

Why Use Events?

- **Loose Coupling**: Publisher and subscriber don't need direct knowledge of each other.
- **Scalability**: Easy to add/remove subscribers without changing publisher.
- **Modularity**: Encapsulates notification logic.

Declaring an Event

- Uses the **`event`** keyword.
- Typically based on a delegate (often `EventHandler` or `EventHandler<TEventArgs>`).
- **Syntax**: `public event [DelegateType] EventName;`
- **Example**:

```
public event EventHandler<OrderProcessedEventArgs>
OrderProcessed;
```

- `event` **keyword protection**: Only declaring class can raise; external classes only add/remove handlers.

Standard Event Pattern in .NET

- **Delegate**: EventHandler` or `EventHandler<TEventArgs>.
  - object sender: The object that raised the event.
  - TEventArgs e : An object derived from EventArgs containing event data.
- **Custom** EventArgs: Create a class inheriting from EventArgs to pass custom data.
- **Example**:

```
public class OrderProcessedEventArgs : EventArgs { /* ... */ }
```

Raising an Event

- Typically done within a **`protected virtual void`** `OnEventName(EventArgs e)` method.
- **Null-conditional operator** (`?.`): Check for subscribers before invoking.
- **Example**:

```
OrderProcessed?.Invoke(this, new OrderProcessedEventArgs(...));
```

Subscribing to an Event

- Use the **+=** operator to attach an event handler method.
- Handler method signature must match the event's delegate.
- **Example**:

```
publisher.OrderProcessed += MyEventHandlerMethod;
```

- **Unsubscribing**: Use **-=** to detach (important for memory management).

Event Demo

- **Scenario**: Order Processing System
- **Publisher**: `OrderProcessor` (raises `OrderProcessed` event).
- **Subscribers**: `EmailService`, `SmsService`, `Logger` (handle the event).
- **Show**: Event declaration, custom `EventArgs`, raising, subscribing, and handling.

Built-in Delegates

- C# provides generic delegates for common scenarios.
- Reduce need for custom delegate declarations.
- `Action`: For methods that return `void`.
- `Func`: For methods that return a value.
- `Predicate`: For methods that return `bool` and take one parameter.

Action Delegates

- **Purpose**: Represents a method that takes parameters but *does not return a value* (**`void`**).
- **Syntax**: `Action<T1, T2, ... >`
- **Example**:
- `Action<string> print = msg => Console.WriteLine(msg);`
- **Demo**: Simple `Action` usage.

Func Delegates

- **Purpose**: Represents a method that takes parameters and *returns a value*.
- **Syntax**:
- Func<T1, T2, ..., TResult>` (last type is return type).
- **Example**: Func<int, int, int> add = (a, b) => a + b;
- **Demo**: Simple **Func** usage, often with LINQ.

`Predicate<T>` Delegate

- **Purpose**: Represents a method that takes one parameter of type `T` and *returns a* ***`bool`*** *value*.
- **Syntax**: `Predicate<T>`
- **Example**:

```
Predicate<int> isEven = num => num % 2 == 0;
```

- **Demo**: Using `Predicate` with `List<T>.FindAll()`.

Day 6 Recap

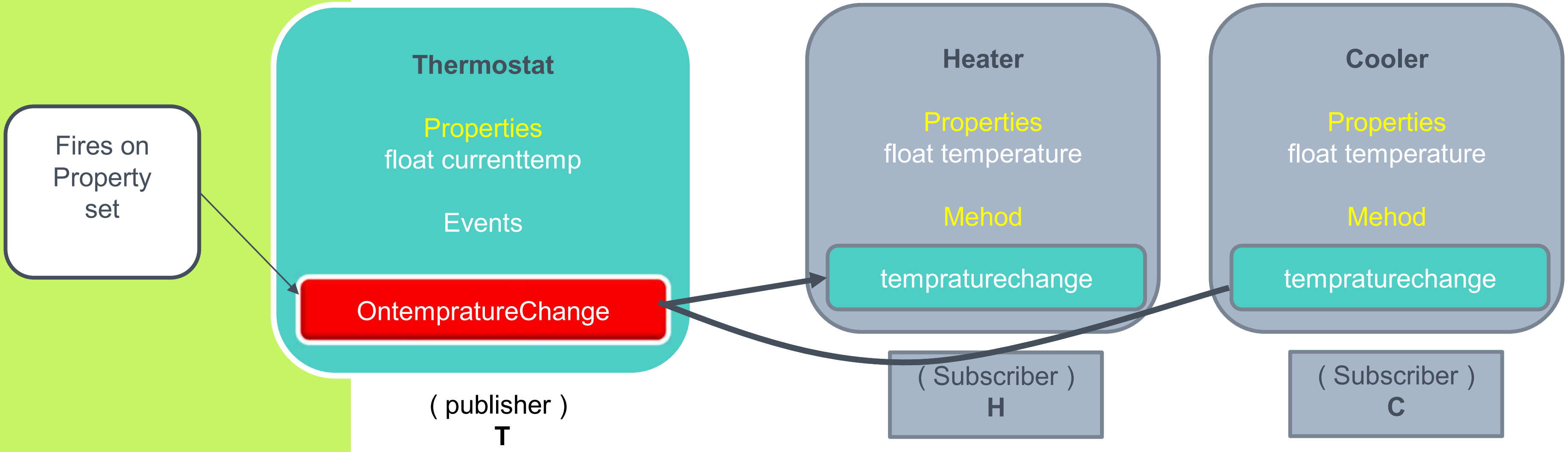
- **Delegates**: Type-safe function pointers, enable callbacks.
- **Multicast Delegates**: Invoke multiple methods.
- **Events**: Publisher-subscriber model, loose coupling, **`event`** keyword.
- **Standard Event Pattern**: `EventHandler<TEventArgs>`.
- **Built-in Delegates**: `Action`, `Func`, `Predicate` for common scenarios.

Q&A and Next Steps

- Questions?
- Tomorrow: Advanced Topics!

Assignment

- Implement and Trace Bubble Sort using delegate
- Write a program for heater , cooler and thermostat



```
T.OntempratureChange += C.tempraturechange;
T.OntempratureChange += H.tempraturechange;
```

Day 7

Advanced Topics

Agenda for Day 7

- Extension Methods
- Anonymous Types
- Lambda Expressions (Review & Advanced Usage)
- Asynchronous Programming (`async`/`await`)
- Attributes

Extension Methods

- **Concept**: Add new methods to existing types without modifying them.
- Syntax:
  - Defined in a `static` class.
  - First parameter preceded by `this` keyword.
- **Example**:

```
public static int WordCount(this string s) {  ...  }
```

- **Usage**: Call as if it's an instance method:

```
  myString.WordCount();
```

Benefits of Extension Methods

- **Readability**: Fluent API style.
- **Reusability**: Centralize utility methods.
- **Extensibility**: Add functionality to `sealed` classes or types you don't own.
- **LINQ**: All LINQ query operators are extension methods.

Anonymous Types

- **Concept**: Compiler-generated types for temporary, read-only data structures.
- **Characteristics**:
  - No explicit class definition.
  - Properties are read-only.
  - Type name generated by compiler (not accessible in code).
- **Syntax**: `var person = new { Name = "Alice", Age = 30 };`
- **Use Cases**: Primarily with LINQ projections.

Anonymous Types Example

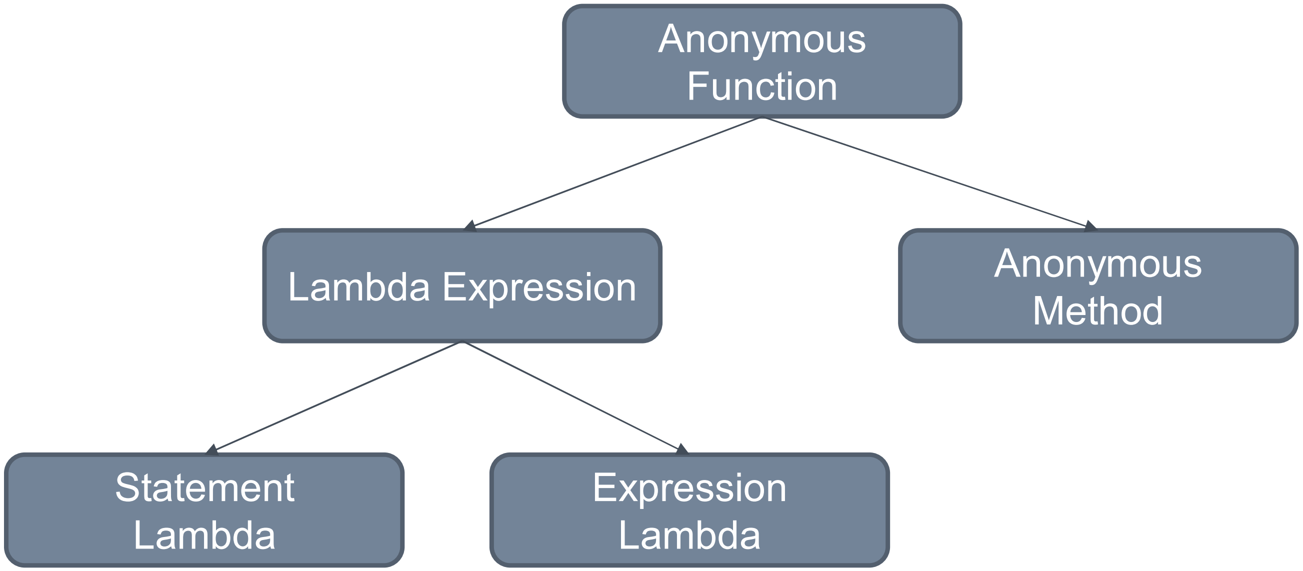
- **Code Example**:

```
var products = new[] {
    new { Name = "Laptop", Price = 1200 },
    new { Name = "Mouse", Price = 25 }
};
var highPriced = products.Where(p => p.Price > 100)
                         .Select(p => new { p.Name, p.Price });
```

- **Limitations**: Cannot be passed across method boundaries easily.

Anonymous Function

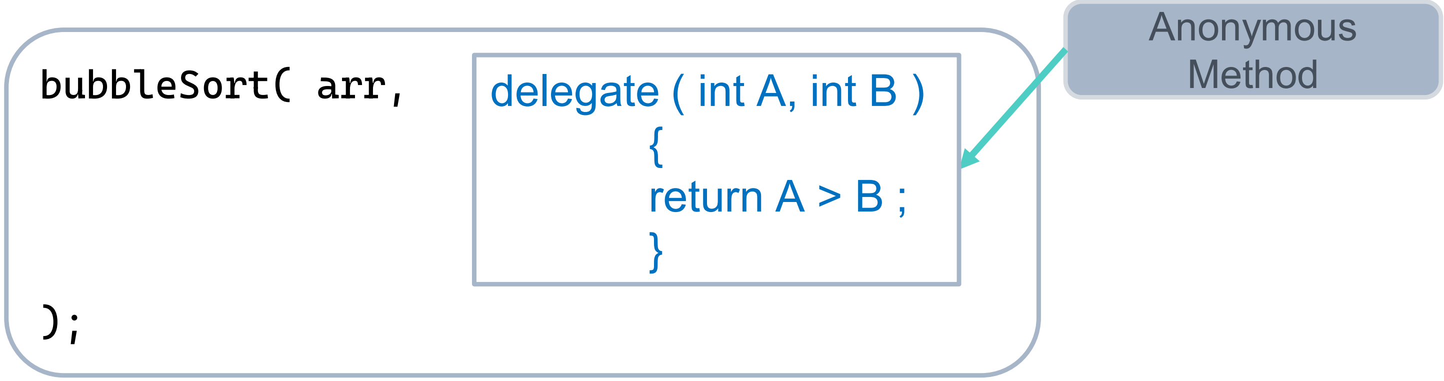
- An anonymous function is an "inline" statement or expression that can be used wherever a **delegate type is expected**



Anonymous Method

```
public delegate bool Mydelegate(int L, int M); //Declare Delegate

void bubbleSort(int[] arr, Mydelegate d) //Declare Method
{  ...}
```



```
bubbleSort( arr,   delegate ( int A, int B )
                            {
                            return A > B ;
                            }

);
```

Anonymous Method

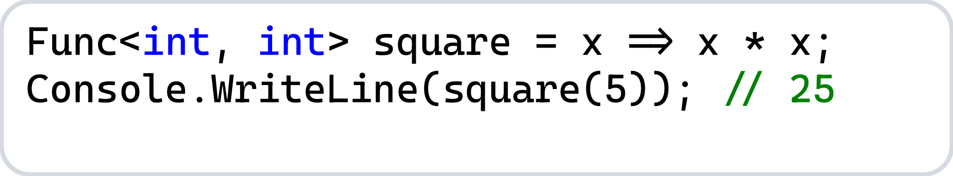
Anonymous Method

- anonymous method could access outer method variables (like local method)

```
void method1()
        {
            string str ="Hello";
            method2(delegate (string s)
              {
                Console.writline(str +" " +s);
              }
            );
        }
void mehod2(Action<string> action)
        {
            action("world");
        }
```

Lambda Expression

- Code could be represented as Expression that compiled to delegate
- Delegate Types
  - Action delegate →no return
  - Func delegate → return
- Lambda expression anatomy



```
Func<int, int> square = x => x * x;
Console.WriteLine(square(5)); // 25
```

Lambda Expression

- Statement Lambda

```
bubbleSort(arr,
       (A,B) =>
       {
          return A > B;
       }
);
```

  - One parameter

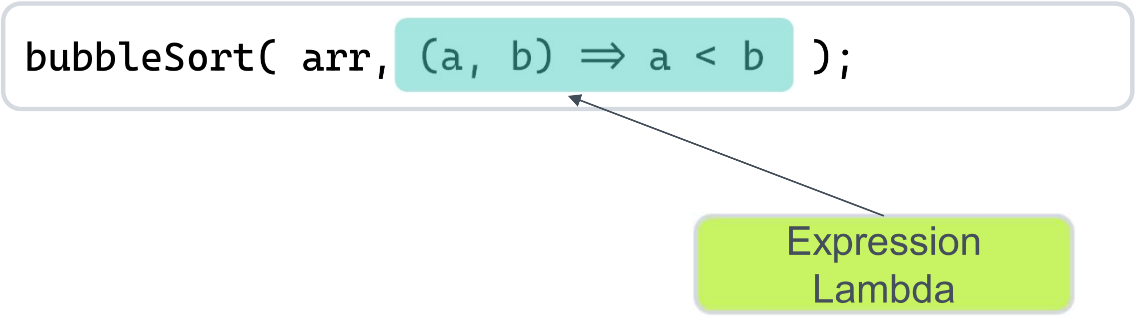
```
A=>
```

  - Parameterles

```
()=>
```

Lambda Expression

- Expression Lambda



```
bubbleSort( arr, (a, b) => a < b );
```

Expression Lambda

Lambda Expressions (Review)

- **Concept**: Concise way to write anonymous methods.
- **Syntax**: `parameters ⇒ expression` or
  `parameters ⇒ {statements; }`
- **Benefits**: Conciseness, readability, flexibility.
- **Use Cases**: LINQ, Delegates (`Action`, `Func`, `Predicate`), Event Handlers.

Lambda Expressions: Advanced Usage

- **Closures**: Lambdas can "capture" variables from their surrounding scope.
- **Example**:

```
int factor = 2;
Func<int, int> multiply = x => x * factor; // 'factor' is captured
Console.WriteLine(multiply(5)); // 10
```

- **Caution**: Be aware of potential side effects with captured variables.

Asynchronous Programming: The Problem

- **Responsiveness**: Long-running operations (network, file I/O) block the main thread, freezing UI.
- **Scalability**: Blocking threads limits concurrent requests in server apps.
- **Traditional Solutions**: Complex callbacks, `BackgroundWorker`.

`async` and `await` : The Solution

- **Concept**: Keywords that simplify asynchronous code.
- **`async` keyword**: Marks a method as asynchronous; allows `await` inside.
- **`await` keyword**: Pauses execution of the **`async`** method until the awaited **`Task`** completes.
- **Crucially**: Does **not** block the calling thread.

async / await Flow

- When `await` is encountered:
  - Control returns to the caller.
  - Calling thread remains unblocked.
  - When awaited Task completes, execution resumes in the async method.
- **Return Types**: Task, Task<TResult>, void (avoid `void` unless event handler).

async / await Example

- **Code Example**:

```
public async Task<string> FetchDataAsync()
{
    Console.WriteLine("Fetching data...");
    await Task.Delay(3000); // Simulate network call
    return "Data fetched!";
}

// In Main:
string result = await FetchDataAsync();
Console.WriteLine(result);
```

- **Benefits**: Simplicity, Responsiveness, Performance