### 1. **Reflection: Inspecting Types**

- **Exercise**: Create a method that accepts any object and prints out its type name and all its methods.

- **Solution**:

```csharp
public void PrintTypeInfo(object obj)
{
    Type type = obj.GetType();
    Console.WriteLine("Type: " + type.Name);
    MethodInfo[] methods = type.GetMethods();
    foreach (var method in methods)
    {
        Console.WriteLine("Method: " + method.Name);
    }
}
```

### 2. **Reflection: Invoking a Method**

- **Exercise**: Use reflection to invoke a private method in a class.

- **Solution**:

  ```csharp
  public class MyClass
  {
      private void MyPrivateMethod()
      {
          Console.WriteLine("Private Method Invoked");
      }
  }

  public void InvokePrivateMethod()
  {
      MyClass myClass = new MyClass();
      MethodInfo method = typeof(MyClass).GetMethod("MyPrivateMethod", BindingFlags.NonPublic | BindingFlags.Instance);
      method.Invoke(myClass, null);
  }
  ```

### 3. **Reflection: Working with Attributes**

- **Exercise**: Create a custom attribute and apply it to a class. Then, retrieve and print the attribute value using reflection.

- **Solution**:

```csharp
[AttributeUsage(AttributeTargets.Class)]
public class MyCustomAttribute : Attribute
{
    public string Description { get; }
    public MyCustomAttribute(string description) => Description = description;
}

[MyCustomAttribute("This is a test class.")]
public class TestClass {}

public void PrintAttribute()
{
    Type type = typeof(TestClass);
```

```
        var attribute =
(MyCustomAttribute)Attribute.GetCustomAttribute(
type, typeof(MyCustomAttribute));
        Console.WriteLine(attribute.Description);
    }
    ```


### 4. **Delegates: Basic Delegate Usage**

  - **Exercise**: Declare a delegate that accepts an integer and returns the square of that integer. Write a method that takes this delegate as a parameter.

  - **Solution**:

    ```csharp
    public delegate int SquareDelegate(int x);


    public int Square(int x) => x * x;


    public void ExecuteDelegate(SquareDelegate del,
int value)
    {
        int result = del(value);
        Console.WriteLine("Result: " + result);
```

```
}
```

### 5. **Delegates: Multicast Delegate**

- **Exercise**: Create a multicast delegate that points to two methods, one that adds two numbers and another that subtracts them. Invoke the delegate.

- **Solution**:

```csharp
public delegate void MathDelegate(int a, int b);

public void Add(int a, int b) => Console.WriteLine("Add: " + (a + b));
public void Subtract(int a, int b) => Console.WriteLine("Subtract: " + (a - b));

public void ExecuteMulticastDelegate()
{
    MathDelegate mathDel = Add;
    mathDel += Subtract;
    mathDel(5, 3);
```

```
}
```

### 6. **Anonymous Delegates**

- **Exercise**: Use an anonymous delegate to find and print the maximum of two numbers.
- **Solution**:

```csharp
public delegate int MaxDelegate(int a, int b);

public void FindMax()
{
    MaxDelegate maxDel = delegate(int a, int b)
    {
        return a > b ? a : b;
    };
    Console.WriteLine("Max: " + maxDel(10, 20));
}
```

### 7. **Covariance in Delegates**

- **Exercise**: Demonstrate covariance by assigning a method returning a string to a delegate expecting a method that returns an object.
- **Solution**:

```csharp
public delegate object MyCovariantDelegate();

public string GetString() => "Hello, Covariance!";

public void DemonstrateCovariance()
{
    MyCovariantDelegate del = GetString;
    Console.WriteLine(del());
}
```

### 8. **Contravariance in Delegates**

- **Exercise**: Demonstrate contravariance by assigning a method accepting a base class parameter to a delegate expecting a derived class parameter.
- **Solution**:

```csharp
```

```
    public class BaseClass {}
    public class DerivedClass : BaseClass {}

    public delegate void
MyContravariantDelegate(DerivedClass d);

    public void ProcessBaseClass(BaseClass b) =>
Console.WriteLine("Processing BaseClass");

    public void DemonstrateContravariance()
    {
        MyContravariantDelegate del =
ProcessBaseClass;
        del(new DerivedClass());
    }
    ```

### 9. **Async Callbacks**
  - **Exercise**: Create an asynchronous operation that performs a long-running task and invokes a callback method upon completion.
  - **Solution**:

```csharp
public delegate void Callback(int result);

public void LongRunningTask(Callback callback)
{
    Task.Run(() =>
    {
        Thread.Sleep(2000); // Simulate long task
        callback(42);
    });
}

public void TaskCompleted(int result)
{
    Console.WriteLine("Task completed with result: " + result);
}

public void ExecuteAsyncTask()
{
    LongRunningTask(TaskCompleted);
```

```
}
```

### 10. **Custom Events**

- **Exercise**: Create a custom event that triggers when a threshold value is exceeded.

- **Solution**:

```csharp
public delegate void ThresholdExceededEventHandler(object sender, EventArgs e);


public class ThresholdNotifier
{
    public event ThresholdExceededEventHandler ThresholdExceeded;


    private int _threshold = 10;
    public void CheckValue(int value)
    {
        if (value > _threshold)
        {
```

```
            OnThresholdExceeded(EventArgs.Empty);
        }
    }

    protected virtual void OnThresholdExceeded(EventArgs e)
    {
        ThresholdExceeded?.Invoke(this, e);
    }
}

public void HandleThresholdExceeded(object sender, EventArgs e)
{
    Console.WriteLine("Threshold exceeded!");
}

public void TestCustomEvent()
{
    ThresholdNotifier notifier = new ThresholdNotifier();
```

```
        notifier.ThresholdExceeded +=
HandleThresholdExceeded;
        notifier.CheckValue(15);
    }
    ```


### 11. **Creating and Starting a Thread**
  - **Exercise**: Create a thread that prints numbers from 1 to 10.
  - **Solution**:
    ```csharp
    public void PrintNumbers()
    {
        for (int i = 1; i <= 10; i++)
        {
            Console.WriteLine(i);
        }
    }

    public void StartThread()
    {
```

```
        Thread thread = new Thread(PrintNumbers);
        thread.Start();
    }
    ```

### 12. **Thread Priority**
  - **Exercise**: Create two threads with different priorities and observe the order of execution.
  - **Solution**:
    ```csharp
    public void HighPriorityTask()
    {
        for (int i = 0; i < 5; i++)
        {
            Console.WriteLine("High Priority Task");
        }
    }

    public void LowPriorityTask()
    {
        for (int i = 0; i < 5; i++)
```

```
        {
            Console.WriteLine("Low Priority Task");
        }
    }

    public void SetThreadPriority()
    {
        Thread highPriorityThread = new
Thread(HighPriorityTask);
        Thread lowPriorityThread = new
Thread(LowPriorityTask);

        highPriorityThread.Priority =
ThreadPriority.Highest;
        lowPriorityThread.Priority =
ThreadPriority.Lowest;

        highPriorityThread.Start();
        lowPriorityThread.Start();
    }
    ```
```
```

### 13. **Interrupting a Thread**

- **Exercise**: Create a thread that waits indefinitely until it is interrupted.

- **Solution**:

```csharp
public void WaitIndefinitely()
{
    try
    {
        Thread.Sleep(Timeout.Infinite);
    }
    catch (ThreadInterruptedException)
    {
        Console.WriteLine("Thread was interrupted!");
    }
}

public void InterruptThread()
{
    Thread thread = new Thread(WaitIndefinitely);
```

```
        thread.Start();
        Thread.Sleep(2000);
        thread.Interrupt();
    }
    ```

### 14. **Background vs Foreground Threads**
  - **Exercise**: Demonstrate the difference between background and foreground threads by creating one of each type and observing application exit behavior.
  - **Solution**:
    ```csharp
    public void BackgroundTask()
    {
        Console.WriteLine("Background task starting...");
        Thread.Sleep(5000);
        Console.WriteLine("Background task completed.");
    }

    public void ForegroundTask()
```

```
    {
        Console.WriteLine("Foreground task starting...");
        Thread.Sleep(5000);
        Console.WriteLine("Foreground task completed.");
    }

    public void TestThreadTypes()
    {
        Thread backgroundThread = new Thread(BackgroundTask) { IsBackground = true };
        Thread foregroundThread = new Thread(ForegroundTask);

        backgroundThread.Start();
        foregroundThread.Start();
    }
    ```

### 15. **Using the Thread Pool**

- **Exercise**: Queue a simple task to the thread pool and display the thread's managed ID.
- **Solution**:

```csharp
public void ThreadPool

Task(object state)
    {
        Console.WriteLine("Thread Pool Task running on thread: " + Thread.CurrentThread.ManagedThreadId);
    }

    public void QueueTaskToThreadPool()
    {

ThreadPool.QueueUserWorkItem(ThreadPoolTask);
    }
```

### 16. **Synchronization Using Monitor**

- **Exercise**: Create a scenario where multiple threads update a shared resource. Use `Monitor` to ensure thread safety.

- **Solution**:

```csharp
private int _counter = 0;
private readonly object _lockObject = new object();


public void IncrementCounter()
{
    for (int i = 0; i < 1000; i++)
    {
        Monitor.Enter(_lockObject);
        try
        {
            _counter++;
        }
        finally
        {
            Monitor.Exit(_lockObject);
```

```
        }
    }
}

public void TestMonitor()
{
    Thread t1 = new Thread(IncrementCounter);
    Thread t2 = new Thread(IncrementCounter);

    t1.Start();
    t2.Start();

    t1.Join();
    t2.Join();

    Console.WriteLine("Final Counter Value: " +
_counter);
}
```

### 17. **Synchronization Using Mutex**

- **Exercise**: Create a program where two threads access a shared resource across processes. Use a `Mutex` to synchronize access.
- **Solution**:

```csharp
private Mutex mutex = new Mutex();


public void AccessResource()
{
    mutex.WaitOne();
    try
    {
        Console.WriteLine("Thread " + Thread.CurrentThread.ManagedThreadId + " is accessing the resource");
        Thread.Sleep(2000); // Simulate resource access
    }
    finally
    {
        mutex.ReleaseMutex();
    }
```

```
    }

    public void TestMutex()
    {
        Thread t1 = new Thread(AccessResource);
        Thread t2 = new Thread(AccessResource);

        t1.Start();
        t2.Start();

        t1.Join();
        t2.Join();
    }
    ```

### 18. **Using `lock` Statement**
  - **Exercise**: Implement a thread-safe counter using the `lock` statement.
  - **Solution**:
    ```csharp
    private int _safeCounter = 0;
```

```
    private readonly object _lockObject = new
object();

    public void IncrementSafeCounter()
    {
       lock (_lockObject)
       {
          _safeCounter++;
       }
    }

    public void TestLockStatement()
    {
       Thread t1 = new
Thread(IncrementSafeCounter);
       Thread t2 = new
Thread(IncrementSafeCounter);

       t1.Start();
       t2.Start();
```

```
        t1.Join();
        t2.Join();

        Console.WriteLine("Final Safe Counter Value: " + _safeCounter);
    }
    ```

### 19. **Using Semaphore**
  - **Exercise**: Create a program where multiple threads attempt to access a limited resource. Use a `Semaphore` to restrict access to a specific number of threads.
  - **Solution**:
    ```csharp
    private Semaphore semaphore = new Semaphore(2, 2); // Max 2 threads

    public void AccessLimitedResource()
    {
        semaphore.WaitOne();
        try
```

```
        {
            Console.WriteLine("Thread " +
Thread.CurrentThread.ManagedThreadId + " is
accessing the limited resource");
            Thread.Sleep(3000); // Simulate resource
access
        }
        finally
        {
            semaphore.Release();
        }
    }

    public void TestSemaphore()
    {
        for (int i = 0; i < 5; i++)
        {
            Thread t = new
Thread(AccessLimitedResource);
            t.Start();
        }
    }
```

```

### 20. **Using ManualResetEvent**

- **Exercise**: Implement a scenario where one thread waits for a signal from another thread using `ManualResetEvent`.

- **Solution**:

```csharp
private ManualResetEvent manualResetEvent = new ManualResetEvent(false);

public void WaitingThread()
{
    Console.WriteLine("Waiting for signal...");
    manualResetEvent.WaitOne();
    Console.WriteLine("Signal received, proceeding...");
}

public void SignalingThread()
{
    Console.WriteLine("Sending signal...");
```
```

```
    Thread.Sleep(2000); // Simulate some work
    manualResetEvent.Set();
}

public void TestManualResetEvent()
{
    Thread t1 = new Thread(WaitingThread);
    Thread t2 = new Thread(SignalingThread);

    t1.Start();
    t2.Start();

    t1.Join();
    t2.Join();
}
```