Object Oriented Programming

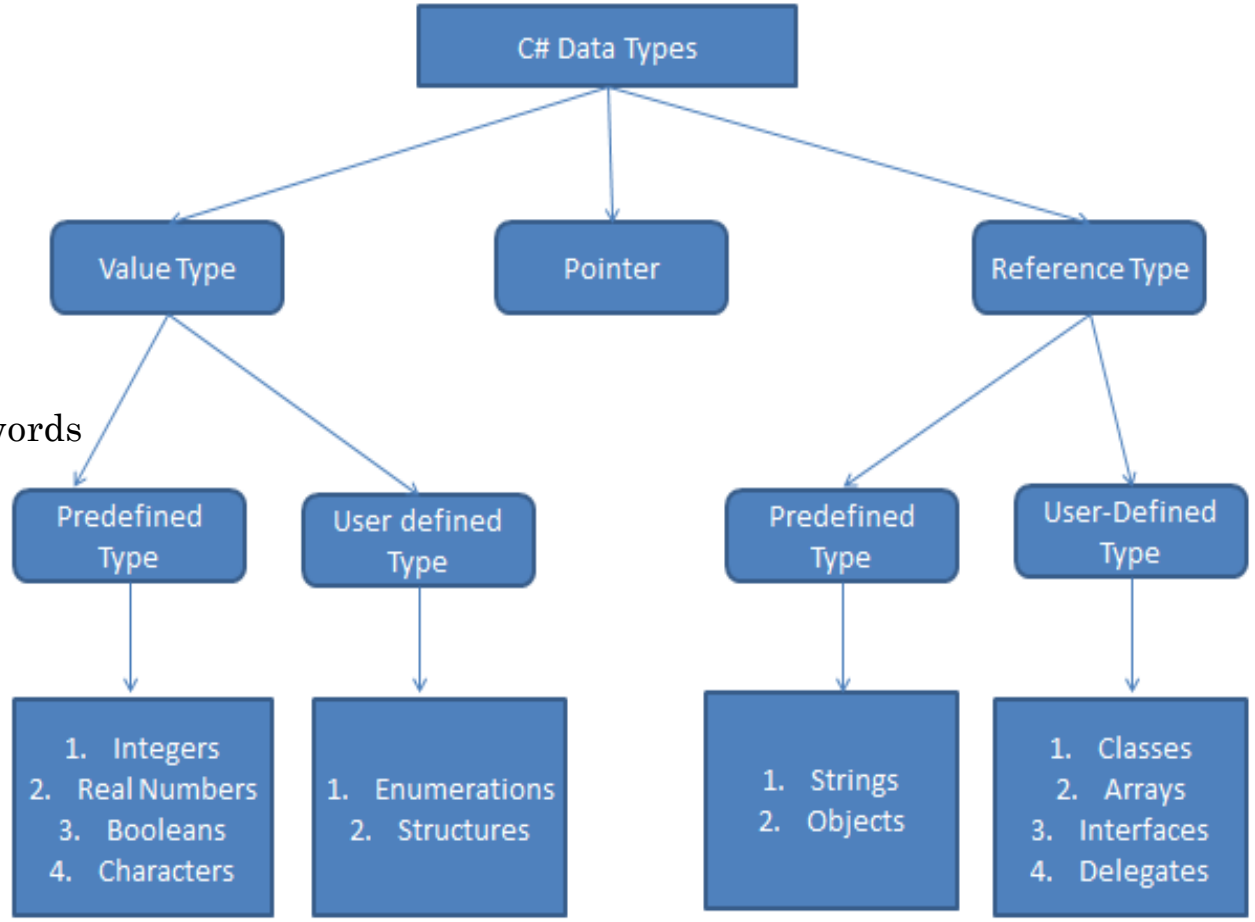
with .Net Core, C#

Prof. Dr. Aydın SEÇER

contents

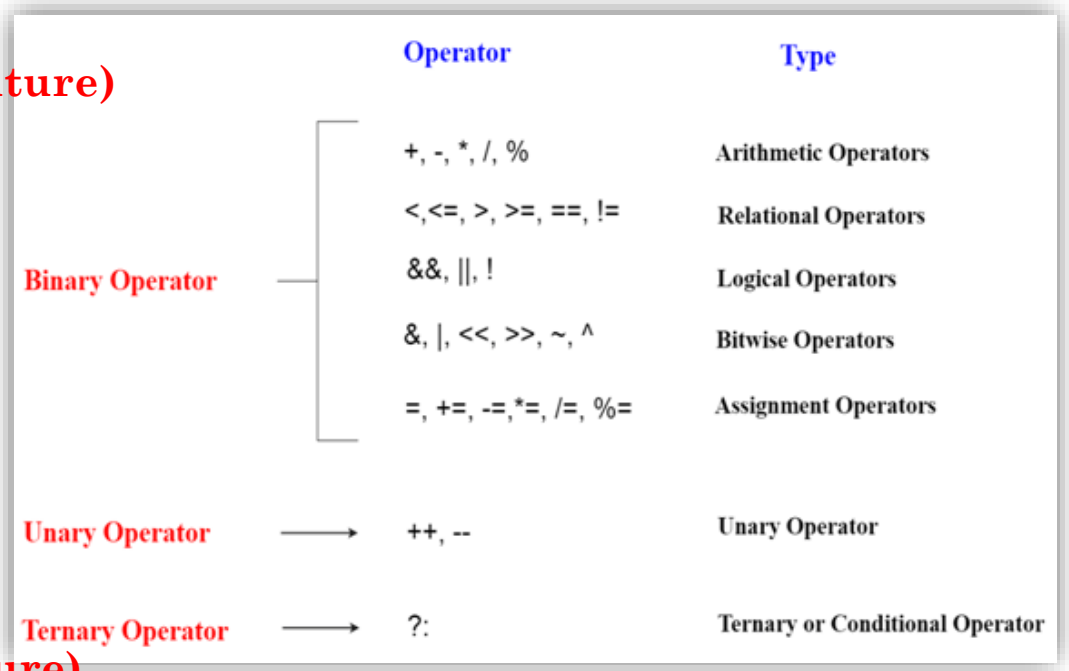
Basics and Data Types-1

- Structure Of A C# Program
- Keywords
- What is Re-Sharper
- Data Types
- Variables and their definitons
- **var (anonymous)** and **dynamic** keywords
- Constants
- Readonly keyword
- Overflow
- Scope Rule
- Comments
- Input and output methods



Basics and Data Types-2

- Type Conversion
  - Imlicit
  - Explicit
- Boxing and Unboxing- (**will be discussed in Future)**
- Operators:
  - Arithmetic
  - Relational
  - Logical
  - Bitwise
  - Assingment
  - Unary
  - Ternary, conditional
- Operator overloading (**will be discussed in Future)**

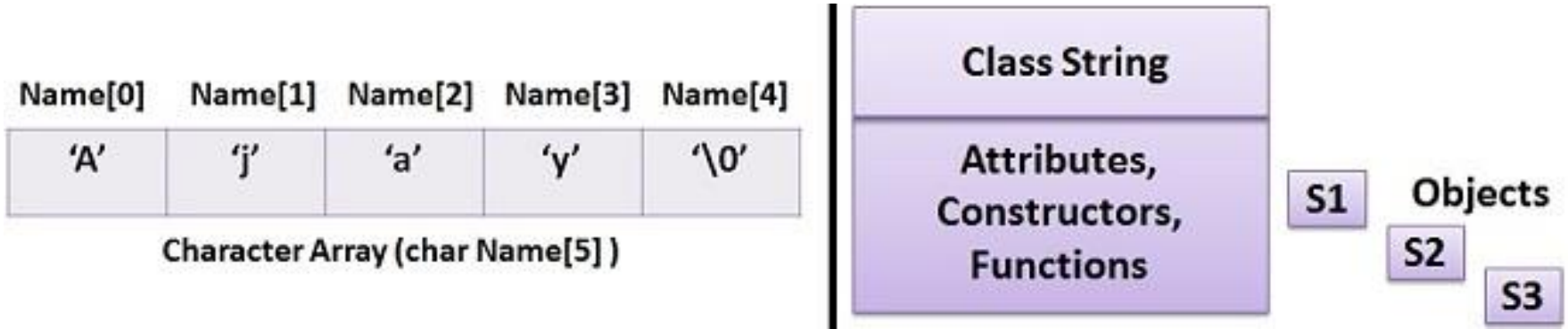


Basics and Data Types-3

- **«sizeof()», «typeof()», «is» and «as» operatosArray Basics**
- **Arrays of primitive types,**
- **2D and higher dimensional arrays**
- **NULL type (‘\0’ macro in C, Null Reference in C#, only reference types can be null)**
- **Conditional Statements**
  - if, if…else if, else
  - switch, case
  - Conditional if, val1==val2 ? if-true : if-false
- **Control Flows and Loops**
  - for,
  - foreach
  - while
  - Do.. while
  - goto,
  - Infinite loops
  - Recursive functions (**will be discussed in Future)**
- **break and continue keywords.**

String Type

- String in memory
- String is immutable
- String is reference type, string is a class
- Char Array vs String
- Char Array is Mutable
- String Comparision, S
- String arrays
- Some useful string methods, String.Format() method, @ and $ symbol usage in strings
- StringBuilder Class (will be discussed in Future)



Methods (Functions) in OOP

- Defining a method
- Method Types according to signature
  - void MethodName(void)
  - type MethodName(void)
  - type MethodName(type)
  - void MethodName(type)
- Constructor methods (will be discussed in Future)
- Destructor (finalizer) methods (will be discussed in Future)
- Extension Methods (will be discussed in Future)
- Constructor methods (will be discussed in Future)
- Asyncronous Methods (will be discussed in Future)
- Anonymous Methods (will be discussed in Future)
- Optional Parameters in methods: MethodName (int x =1)
- Methods with variable number parameters: MethodName (string[] args) or int[] args…
- Calling methods with random order parameters MethodName(x:1,y:2)
- Methods which can return object(will be discussed in Future)
- Methods which can get object parameters (will be discussed in Future)
- Method overloading
- Methods which arguments are methods/callback (advanced) (will be discussed in Future)

Some useful built-in classes, structs

- Random class
- DateTime struct
- TimeSpan struct
- Math class
- Convert class

Grouping Code Blocks

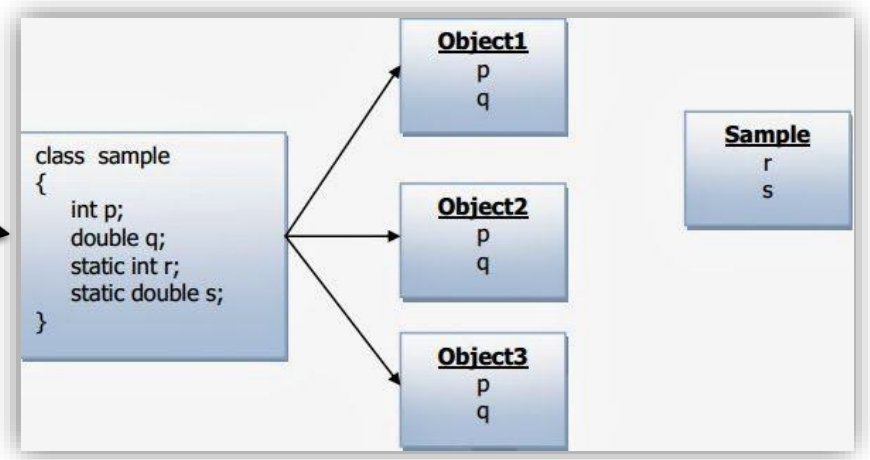
- #region ….#endregion blocks

Introduction to Object Oriented Programming

OOP

Class and class members

- **Class definition**
  - Class is a template for object definition
  - Provides details of any real world object
- **Class Members**
  - Fields
  - Properties
  - Methods
  - Constructor methods: default and overloaded constructors
  - Destructor method : ~classname()
  - Indexers
- **Creating (getting instance) object by new keyword**
- **Object initializers**
- **Constructor methods and new keyword relation**
- **Instance members vs static members**
- **Garbage collector**
  - Finalize() method
  - Dispose() method
  - ~classname() method



Class vs Object

- Class is definiton or template of any object
- Class does not mean object directly (object != class)
- Object is created from class
- Created by using new «new» keyword
- new keyword calls class constructor methods
- Constructor method creates object and store it in the memory
- Object is a variable of class (class members)
- Object is an instance of class

OOP Concepts-1 Encapsulation

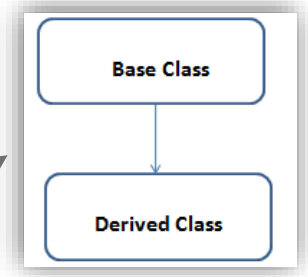
- **Encapsulation**
  - Data hiding
  - Data packaging
  - Capsule of class members for visibility
  - Capsule like medicine tablet to protect members from outside the world
- **Access modifiers for encapsulation**
  - Public (Accessed From Everywhere)
  - Protected (Accessed From Base and Derived Class)
  - Private (Accessed From Class itself)
  - Internal (Accessed From Same Assembly)
  - Protected internal (Accessed from same assebmly and derived class)
- **Access modifiers usage**
  - All class members
  - Classes
  - Interfaces
  - Structs
  - Enumerators

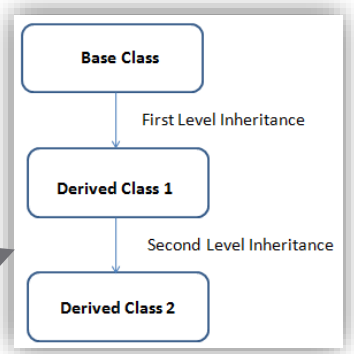
Access Modifiers

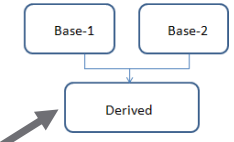
	private	protected	internal	protected internal	public
Within class	Y	Y	Y	Y	Y
Same assembly any other class	N	N	Y	Y	Y
Same assembly other derived classes	N	Y	Y	Y	Y
Outside assembly any other class	N	N	N	N	Y
Outside assembly other derived classes	N	Y	N	Y	Y

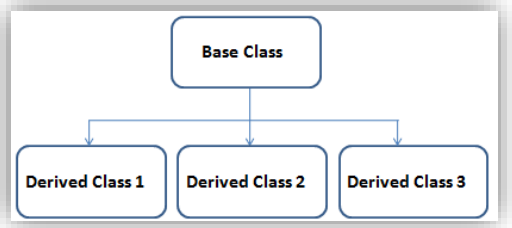
OOP Concepts-2 Inheritance

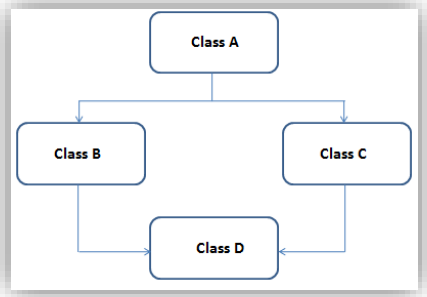
- Inheritance
- Application of Inheritance
- Access Modifiers and Inheritance
- Virtual Methods (can be overriden in children)
- Hide methods and new keyword
- Inhertiance and Overloaded Constructors
- Inheritance Types
  - Single Inheritance
  - Multilevel Inheritance
  - Hierarchical Inheritance
  - Multiple Inheritance (Through Interface)
  - Hybrid Inheritance(Through Interface)











OOP Concepts-3 Abstraction

- **Interfaces**
  - **Is data contract for objects**
  - **Hiding details, only contracts**
  - **It fully abstract**
  - **Is not a class**
  - **Only contains signature of methods,properties, events or fileds**
  - **It can be inherited by only interfaces**
  - **It can be implemented by classes**
- **Inhertiance vs Implemantation**
- **Abstract Classes**
  - **Abstract Methods**
  - **Concrete Methods**
  - **sealed override methods**
  - **Interface vs Abstract Class**
  - **Interface vs Concrete Class**
  - **Abstract Class vs Concrete Class**

- **A class only inherits one base, but can implement multiple interface *****
- **Interfaces supports multiple inheritance from other interfaces *****

OOP Concepts-4 Polymorphism

- Polymorphism
- Virtual methods revisited
- Method override
- Example of polymorphism

Type of Classes

- Regular Class
- Abstract Class
- Static Class
- Sealed Class
- Partial Class

Struct Data Type

- Struct looks and acts like a class
- Struct is value type, not reference type
- Can have, fields, properties, methods and events
- Can implement interfaces
- Can be used without new keyword (But field must be initilized)
- Can be instantiated with new keyword
- Cannot have default constructor but overloaded is possible
- All fields must be intialized from overloaded constructors
- Not support inheritance both from class and struct

Struct Example

- Ref vs value type

```
class Program
{
    0 references
    static void Main(string[] args)
    {

        A a = new A(); a.val1 = 5;
        B b = new B(); b.val1 = 5;
        Console.WriteLine($"a.val1: {a.val1} , b.val1: {b.val1}");
        GetSquare(a,b);
        Console.WriteLine($"a.val1: {a.val1} , b.val1: {b.val1}");
    }
    1 reference
    static void GetSquare(A a, B b)
    {
        a.val1 = a.val1 * a.val1;
        b.val1 = b.val1 * b.val1;
        Console.WriteLine($"a.val1: {a.val1} , b.val1: {b.val1}");
    }
}
3 references
class A
{
    public int val1;
}
3 references
struct B
{
    public int val1;
}
```

Extension Methods, Operator Overloading

- **Extension Methods**
- **Operator Overloading**
- **Anonymous Types**

```
var emp2 = new { EmployeeID = 1, Name = "Demo", Salary = 100000 };
Console.WriteLine(emp2.EmployeeID);
Console.WriteLine(emp2.Name);
```

- **Tuples**

```
var t2 = Tuple.Create(1, 2, 3, 4, 5, 6, 7, Tuple.Create(11, 21, 31));
Console.WriteLine(t2.Rest.Item1.Item1 + " " + t2.Rest.Item1.Item2 + " " +
t2.Rest.Item1.Item3);
```

Operator Overloading Example of Point Class

```
public static Point operator +(Point p1, Point p2)
{
    Point p = new Point();
    p.X = p1.X + p2.X;
    p.Y = p1.Y + p2.Y;
    return p;
}
public static implicit operator string(Point p)
{
    return p.X + ", " + p.Y;
}
public static explicit operator Point(string s)
{
    Point p = new Point();
    p.X = int.Parse(s.Split(',')[0]);
    p.Y = int.Parse(s.Split(',')[1]);
    return p;
}
```

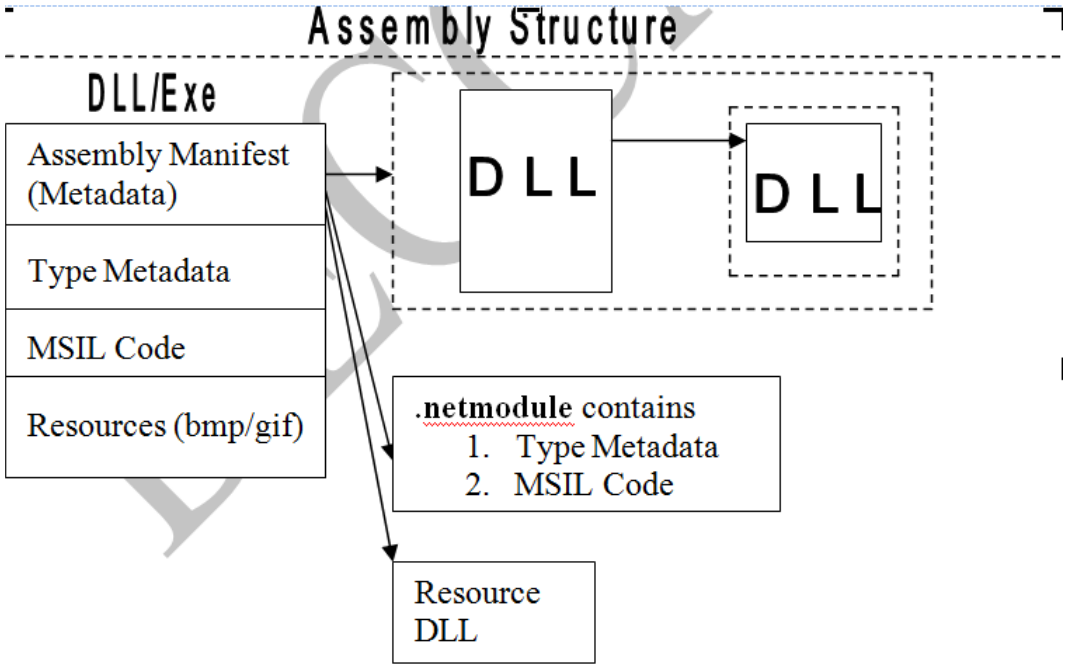
```
Point p1 = new Point(1, 2);
Point p2 = new Point(3, 4);
Point p3 = p1 + p2;
string s = p3; //Implicit Casting from point to string
Console.WriteLine(s);
Point p = (Point)"10,2"; //Explicit Casting
Console.WriteLine(p.X + " " + p.Y);
```

Boxing and Unboxing

- Boxing and Unboxing have performance issue!!!
- Boxing
  - Object o1 = student;
- Unboxing
  - var student1 =(Student)o1

Assemblies and Namespaces

- **Assembly**
- **Namespace**
  - Framework namespaces
  - *Namespaces prevents name conflicts for objects*
  - Aliases
  - .(dot) operator
  - (global::) prevents name conflicts of namespaces.



Framework Defined Useful Interfaces

- INotifyPropertyChanged
- IComparable interface
- IComparable <T> generic interface
- IComparer interface
- IComparer <T> generic interface
- IEquitable <T> generic interface
  - GetHashCode () can be used
- IEqualityComparer <T> generic interface
- IEnumerable interface
  - Can be used with foreach loop for a class that contains array data
- IEnumerator and IEnumerator <T> generic interface
  - Can be used with foreach loop for a class that contains array data
- IEnumerable <T> generic interface
  - Can be used with foreach loop for a class that contains array data
- ICollection interface
  - Can be used with foreach loop for a class that contains array data
- ICollection <T> generic interface
  - Can be used with foreach loop for a class that contains array data

INotifyPropertyChanged

```
class Student:INotifyPropertyChanged
{
    private int _avgGrade;
    3 references
    public int AvgGrage
    {
        get
        {
            return _avgGrade;
        }
        set
        {
            _avgGrade = value;
            OnPropertyChanged(nameof(AvgGrage));
        }
    }

    public event PropertyChangedEventHandler PropertyChanged;

    [NotifyPropertyChangedInvocator]
    1 reference
    protected virtual void OnPropertyChanged([CallerMemberName] string propertyName = null)
    {
        PropertyChanged?.Invoke( sender: this,  e: new PropertyChangedEventArgs(propertyName));
        Console.WriteLine($"Value Changed: {AvgGrage}");
        //send sms, send mail etc...
    }
}
```

IComparable and IComparable <T> generic interface

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        List<Student> students = new List<Student>();
        students.Add( item: new Student() {averageGrade = 100});
        students.Add( item: new Student() { averageGrade = 50 });
        students.Add( item: new Student() { averageGrade = 80 });
        students.Add( item: new Student() { averageGrade = 70 });
        students.Sort();
    }
}
7 references
class Student:IComparable
{
    public int averageGrade=0;
    0 references
    public int CompareTo(object? obj)
    {
        var other = (Student) obj;
        return this.averageGrade.GetHashCode().CompareTo(other.averageGrade.GetHashCode());
    }
}
```

```
class Student:IComparable<Student>
{
    public int averageGrade=0;
    0 references
    public int CompareTo(object? obj)
    {
        var other = (Student) obj;
        return this.averageGrade.GetHashCode().CompareTo(other.averageGrade.GetHashCode());
    }
    0 references
    public int CompareTo(Student other)
    {
        return this.averageGrade.GetHashCode().CompareTo(other.averageGrade.GetHashCode());
    }
}
```

IComparer and IComparer <T> generic interface

```
class Program : IComparer
{
    0 references
    static void Main(string[] args)
    {
        Student student1 = new Student() { averageGrade = 50 };
        Student student2 = new Student() { averageGrade = 75 };

        Program p = new Program();
        Console.WriteLine(p.Compare( x: student1,  y: student2));
    }

    1 reference
    public int Compare( [CanBeNull]object? x,  [CanBeNull]object? y)
    {
        var s1 = (Student)x;
        var s2 = (Student)y;
        return s1.averageGrade.GetHashCode().CompareTo(s2.averageGrade.GetHashCode());
    }
}
6 references
class Student
{
    public int averageGrade = 0;

}
```

```
class Program : IComparer<Student>
{
    0 references
    static void Main(string[] args)
    {
        Student student1 = new Student() { averageGrade = 50 };
        Student student2 = new Student() { averageGrade = 50 };
        Program p = new Program();
        Console.WriteLine(p.Compare(student1, student2));
    }
    1 reference
    public int Compare( [CanBeNull]Student s1,  [CanBeNull]Student s2)
    {
        return s1.averageGrade.GetHashCode().CompareTo(s2.averageGrade.GetHashCode());
    }
}
7 references
class Student
{
    public int averageGrade = 0;
}
```

IEquitable <T> generic interface

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Student student1 = new Student(){averageGrade = 15};
        Student student2 = new Student() { averageGrade = 15 };
        Console.WriteLine(student1.Equals(student2));
    }
}
6 references
class Student : IEquatable<Student>
{
    public int averageGrade = 0;
    1 reference
    public bool Equals([CanBeNull]Student other)
    {
        return this.averageGrade == other.averageGrade;
    }
}
```

IEqualityComparer and IEqualityComparer <T> generic interface

```
class Program:IEqualityComparer
{
    0 references
    static void Main(string[] args)
    {
        Student student1 = new Student(){averageGrade = 15};
        Student student2 = new Student() { averageGrade = 15 };

        Console.WriteLine(new Program().Equals(x:student1,y:student2));
    }

    1 reference
    public bool Equals(object? x, object? y)
    {
        return ((Student)x).GetHashCode() == ((Student)y).GetHashCode();
    }
    0 references
    public int GetHashCode(object obj)
    {
        return ((Student)obj).GetHashCode();
    }
}
7 references
class Student
{
    public int averageGrade = 0;

}
```

```
class Program:IEqualityComparer<Student>
{
    0 references
    static void Main(string[] args)
    {
        Student student1 = new Student(){averageGrade = 15};
        Student student2 = new Student() { averageGrade = 15 };
        Console.WriteLine(new Program().Equals(x:student1,y:student2));
    }
    1 reference
    public bool Equals([CanBeNull]Student x, [CanBeNull]Student y)
    {
        return x.GetHashCode() == y.GetHashCode();
    }
    0 references
    public int GetHashCode([NotNull]Student obj)
    {
        return obj.GetHashCode();
    }
}
8 references
class Student
{
    public int averageGrade = 0;

}
```

IEnumerator and IEnumerator <T> generic interface

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        var student = new Student();
        Console.WriteLine(student.Current);
        while (student.MoveNext())
        {
            Console.WriteLine(student.Current);
        }
    }
}
1 reference
class Student : IEnumerator
{
    public int averageGrade = 0;
    private int current = 0;
    private List<int> list = new List<int>() { 1, 3, 5 };
    2 references
    public object? Current => list[current];
    1 reference
    public bool MoveNext()
    {
        if (list.Count == 0 || list.Count <= current + 1)
            return false;
        current++;
        return true;
    }
    0 references
    public void Reset()
    {
        current = 0;
    }
}
```

```
class Student : IEnumerator<int>
{
    public int averageGrade = 0;
    private int current = 0;

    1 reference
    public bool MoveNext()
    {
        throw new NotImplementedException();
    }

    0 references
    public void Reset()
    {
        throw new NotImplementedException();
    }

    3 references
    public int Current { get; }

    0 references
    object? IEnumerator.Current => Current;

    0 references
    public void Dispose()
    {
    }
}
```

IEnumerable and IEnumerable <T> generic interface

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        var student = new Student();

        foreach (var item :object? in student)
        {
            Console.WriteLine(item);
        }
    }
}
1 reference
class Student : IEnumerable
{
    private List<int> grades = new List<int>() { 1, 3, 5 };

    1 reference
    public IEnumerator GetEnumerator()
    {
        foreach (var item :int in grades)
        {
            yield return item;
        }
    }
}
```

```
class Student : IEnumerable<int>
{
    private List<int> grades = new List<int>() { 1, 3, 5 };
    2 references
    public IEnumerator<int> GetEnumerator()
    {
        foreach (var item :int in grades)
        {
            yield return item;
        }
    }

    0 references
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}
```

ICollection and ICollection <T> generic interface

```
class Student :ICollection
{
    private List<int> grades = new List<int>() { 1, 3, 5 };

    1 reference
    public IEnumerator GetEnumerator()
    {
        foreach (var item:int in grades)
        {
            yield return grades;
        }
    }

    0 references
    public void CopyTo([NotNull]Array array, int index)
    {
      //copy to somewhere
    }

    0 references
    public int Count => grades.Count;
    0 references
    public bool IsSynchronized => true;
    0 references
    public object SyncRoot => null;
}
```

```
class Student
{
    public int Id;
}

1 reference
class StudentList:ICollection<Student>
{
    private List<Student> list;

    0 references
    public StudentList()
    {
        list = new List<Student>();
    }
    1 reference
    public IEnumerator<Student> GetEnumerator()
    {
        foreach (var student in list)
        {
            yield return student;
        }
    }
    0 references
    IEnumerator IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
    0 references
    public void Add([CanBeNull]Student item)
    {
      list.Add(item);
    }
    0 references
    public void Clear()
    {
      list.Clear();
    }
    0 references
    public bool Contains([CanBeNull]Student item)
    {
      return  list.Contains(item);
    }
    0 references
    public void CopyTo([NotNull]Student[] array, int arrayIndex)
    {
        throw new NotImplementedException();
    }
```

Exception Handling

- Try and catch block
- Multiple catch blocks
- Finaly block
- Exception base class
- Exception types
- Custom exceptions
- Throw keyword

IO and Stream

- System.IO.Directory class
- System.IO.File class
- DirectoryInfo Class
- FileInfo Class
- DriveInfo
- Text Reader
- Text Writer
- Binary Reader
- Binary Writer
- Buffered Sream
- Stream Reader
- Stram Writer
- FileStream
- FileSystemInfo
- FileSystemWatcher
- MemoryStream
- Path

Collections

- Nongeneric Collectons (All of them need casting op…)
  - Array List
  - Sorted List / Key based auto sorting
  - Dictionary
  - HashTable
  - Queue
  - Stack
- Generic Collections (No need cast op..)
  - List<T>
  - SortedList<TKey,TValue>
  - Stack<T>
  - Queue<T>
  - Dictionary<Tkey, Tvalue)
  - HashSet<T>
  - SortedSet<T>
  - LinkedList<T>
  - KeyValuePair<TKey, TValue>
  - Custom Tree<T>
  - Custom Graph<T>

Template Types and Methods

- Generic Class
- Generic Method

```
class MySpeacialList<T> where T : class, new ()
{
    private List<T> list;
    0 references
    public MySpeacialList()
    {
        list = new List<T>();
    }
    2 references
    public int Lenght { get; set; }
    0 references
    public void Add(T t)
    {
        Lenght = list.Count;
        list.Add(t);
    }
    0 references
    public void Remove(T t)
    {
        Lenght = list.Count;
        list.Remove(t);
    }
    0 references
    public T Get(int inx)
    {
        return list[inx];
    }
    0 references
    public List<T> GetList()
    {
        return list;
    }
}
```

About Generic Types

- Collections that are type-safe at compile time
- Faster Collection types
- Can be applied for classes and methods as template
- No need BOXING, and UNBOXING operations

Delegates and Events

Delegates

- Delegates are method pointers
- Generic and Nongeneric Delagates
- References methods
- Can be used like variable

```
static void Main(string[] args)
{
    var m1 = new MethodSelector<double>(B);
    m1(t:15);

    DoWork<int>(A,t:5);
}
```

```
public delegate void MehotSelector<T>(T t);
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        DoWork<int>(A,t:5);
    }
    1 reference
    static void DoWork<T>(MehotSelector<T> method,T t)
    {
        method(t);
    }
    1 reference
    static void A(int a)
    {
        Console.WriteLine("A() invoked");
    }
    0 references
    static void B(double b)
    {
        Console.WriteLine("B() invoked");
    }
    0 references
    static void C(float f)
    {
        Console.WriteLine("B() invoked");
    }
}
```

Returning Methods as Parameter

```
public delegate void MethodSelector<T>(T t);
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        GetMethod()(t:4);
    }
    1 reference
    static MethodSelector<int> GetMethod()
    {
        return A;
    }
    1 reference
    static void A(int a)
    {
        Console.WriteLine("A() invoked");
    }
```

Multicast Delagates

```
public delegate void ImageFilter(object photo);
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        ImageFilter filterSet = ApplyFilter1;
        filterSet += ApplyFilter1; //handler method
        filterSet += ApplyFilter2;//handler method
        filterSet.Invoke( photo: "photo file"); //invoke all methods
    }
    2 references
    static void ApplyFilter1(object photo)
    {
        Console.WriteLine("ApplyFilter1 completed...");
    }
    1 reference
    static void ApplyFilter2(object photo)
    {
        Console.WriteLine("ApplyFilter2 completed...");
    }
    0 references
    static void ApplyFilter3(object photo)
    {
        Console.WriteLine("ApplyFilter3 completed...");
    }
}
```

.Net Generic Delegates

- Action
- Func

Action Generic Delegate

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Action<object> m1 = ApplyFilter1;
        Action<object> m2 = ApplyFilter2;
        Action<object> m3 = ApplyFilter3;
        m1( obj: "Image1");
        m2( obj: "Image2");
        m2( obj: "Image3");
    }
    1 reference
    static void ApplyFilter1(object photo)
    {
        Console.WriteLine("ApplyFilter1 completed...");
    }
    1 reference
    static void ApplyFilter2(object photo)
    {
        Console.WriteLine("ApplyFilter2 completed...");
    }
    1 reference
    static void ApplyFilter3(object photo)
    {
        Console.WriteLine("ApplyFilter3 completed...");
    }
}
```

Func Generic Delegate

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Func<object,bool> m1 = ApplyFilter1;
        Func<object, bool> m2 = ApplyFilter2;
        Func<object, bool> m3 = ApplyFilter3;
        m1("Image1");
        m2("Image2");
        m2("Image3");
    }
    1 reference
    static bool  ApplyFilter1(object photo)
    {
        Console.WriteLine("ApplyFilter1 completed...");
        return true;
    }
    1 reference
    static bool ApplyFilter2(object photo)
    {
        Console.WriteLine("ApplyFilter2 completed...");
        return true;
    }
    1 reference
    static bool ApplyFilter3(object photo)
    {
        Console.WriteLine("ApplyFilter3 completed...");
        return true;
    }
}
```

Lambda Expression

- You can write methods
- Everywhere
  - In methods
  - As method parameters

```
class Program
{
    private static Action Method1 = () => { Console.WriteLine("delegate1 executed "); };
    //equals
    //static void Method1()
    //{
    //    Console.WriteLine("delegate1 executed ");
    //}

    private static Action<int> GetSquare = (p) => { Console.WriteLine("delegate1 executed: "+ p*p); };
    //equals
    //static void GetSquare(int p)
    //{
    //    Console.WriteLine("delegate1 executed: "+ p*p);
    //}

    private static Func<int, int, int> Sum = (a, b) => { return   a + b; };
    //equals
    //static int Sum(int a, int b)
    //{
    //    return   a + b;
    //}

    0 references
    static void Main(string[] args)
    {

        Method1();
        GetSquare( obj: 9);
    }
}
```

```
class Program
{
    0 references
    static void Main(string[] args)
    {
        Math( Method: (p :int, q :int)=> { Console.WriteLine(p+q);}, a: 5, b: 8);
    }

    1 reference
    static void Math(Action<int,int> Method,int a, int b )
    {
        Method(a, b);
    }
}
```

Events

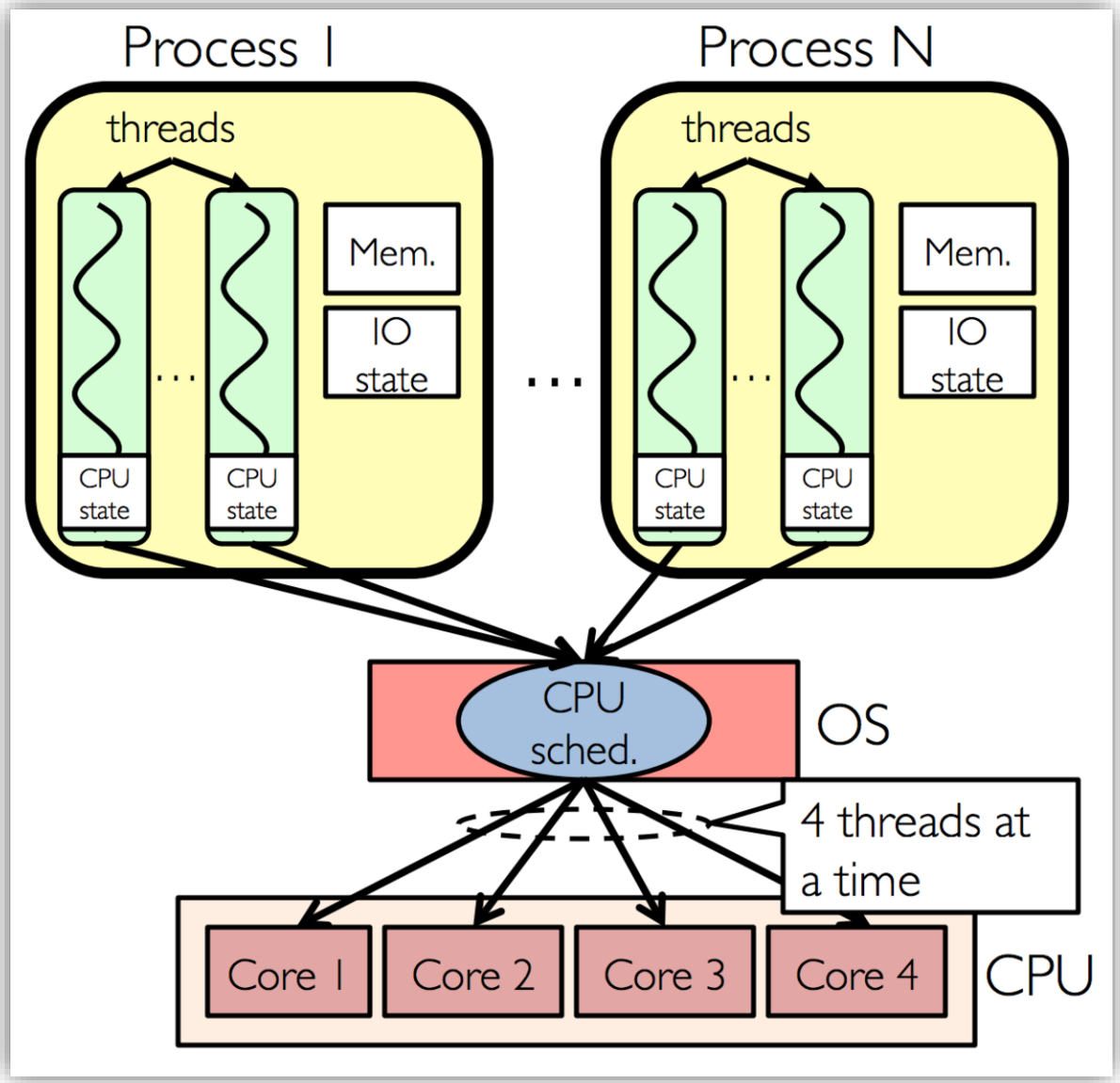
```
class Program
{
    0 references
    static void Main(string[] args)
    {
        DatabaseBackupHelper databaseBackupHelper = new DatabaseBackupHelper();
        databaseBackupHelper.BackupCompleted += DatabaseBackupHelper_BackupCompleted;
        databaseBackupHelper.BackupDatabase();
    }
    1 reference
    private static void DatabaseBackupHelper_BackupCompleted(object sender, EventArgs e)
    {
        Console.WriteLine("Notification Email has been sent to Admin .....");
        //do other staff here log etc
    }
}
2 references
class DatabaseBackupHelper
{
    public delegate void BackupCompletedHandler(object sender, EventArgs e);
    public event BackupCompletedHandler BackupCompleted;
    1 reference
    public void BackupDatabase()
    {
        Console.WriteLine("Backup Started...");
        Thread.Sleep( millisecondsTimeout: 3000);
        Console.WriteLine("Backup In Progress...");
        Thread.Sleep( millisecondsTimeout: 3000);
        Console.WriteLine("Backup Completed...");

        FireEvent();
    }

    1 reference
    private void FireEvent()
    {
        if (BackupCompleted != null)
            BackupCompleted( sender: this, e: EventArgs.Empty);
    }
}
```

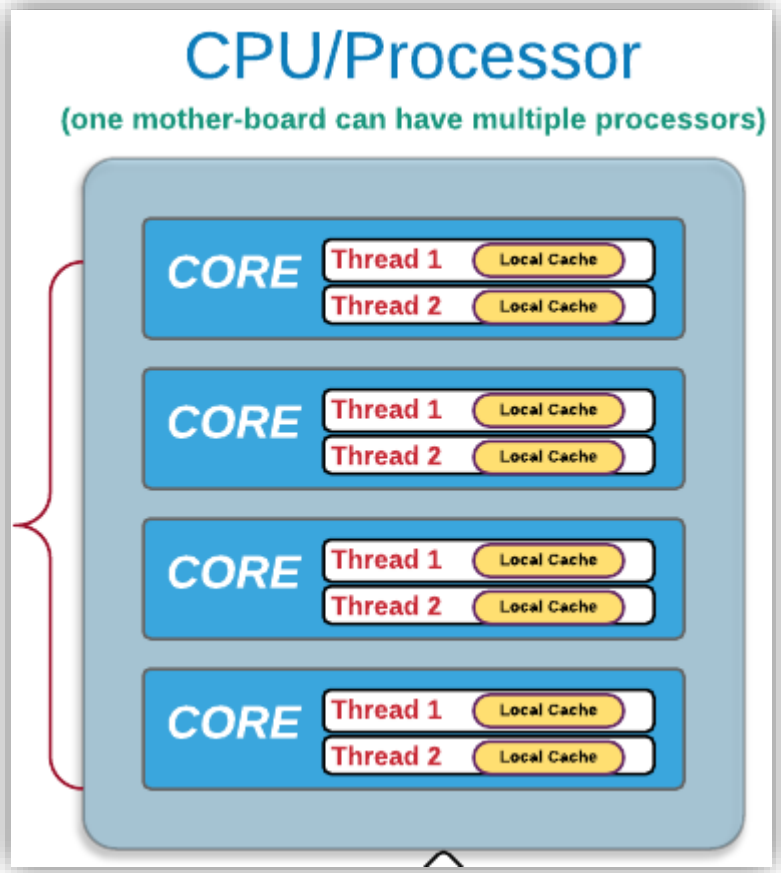
Add or Remove Event Bindings

- instanceObject.someEvent += anyhandler; //ADD handler
- instanceObject.someEvent -= anyhandler; //REMOVE handler

CPU
Process
Multi Threading

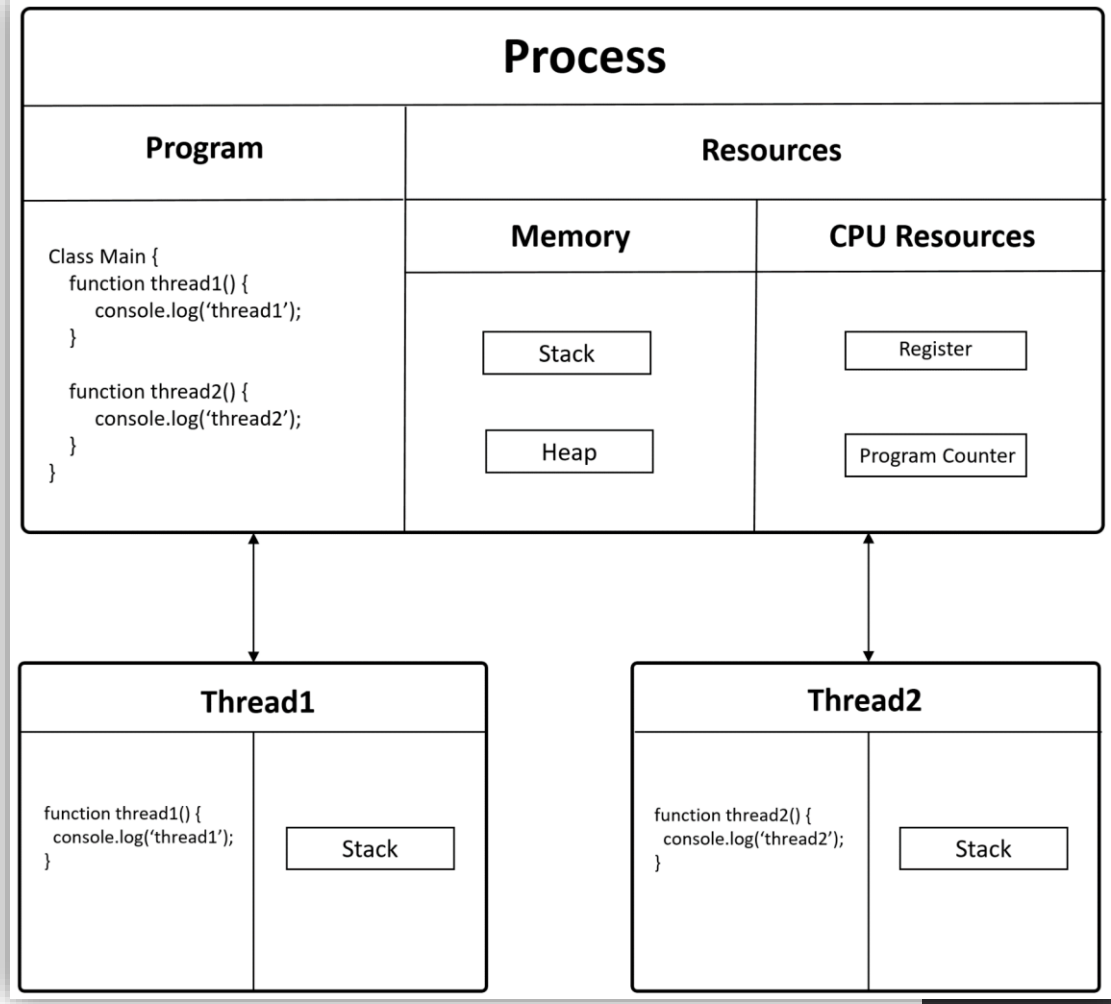
CPU, Process and Threads





Process

- A running instance of any program
- Can be seen from task manager in Windows
- Every process has its own memory address space (execution environment) and it is in that all the data and the instructions of that process reside.
- A process has a self-contained execution environment.
- A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space



Threads

- The path of execution of instructions within a process is called a thread.
- It has a starting point, execution sequence and terminating point.
- Thread is a specific work piece or task piece
- Independent execution path in a program
- Used to perform multiple tasks at the same time as parallel, or asyncronous
- Single Threaded process
  - One thread
- Multi Threaded process
  - Multiple thread
- Main Thread of a program, every program must have at least one thread.

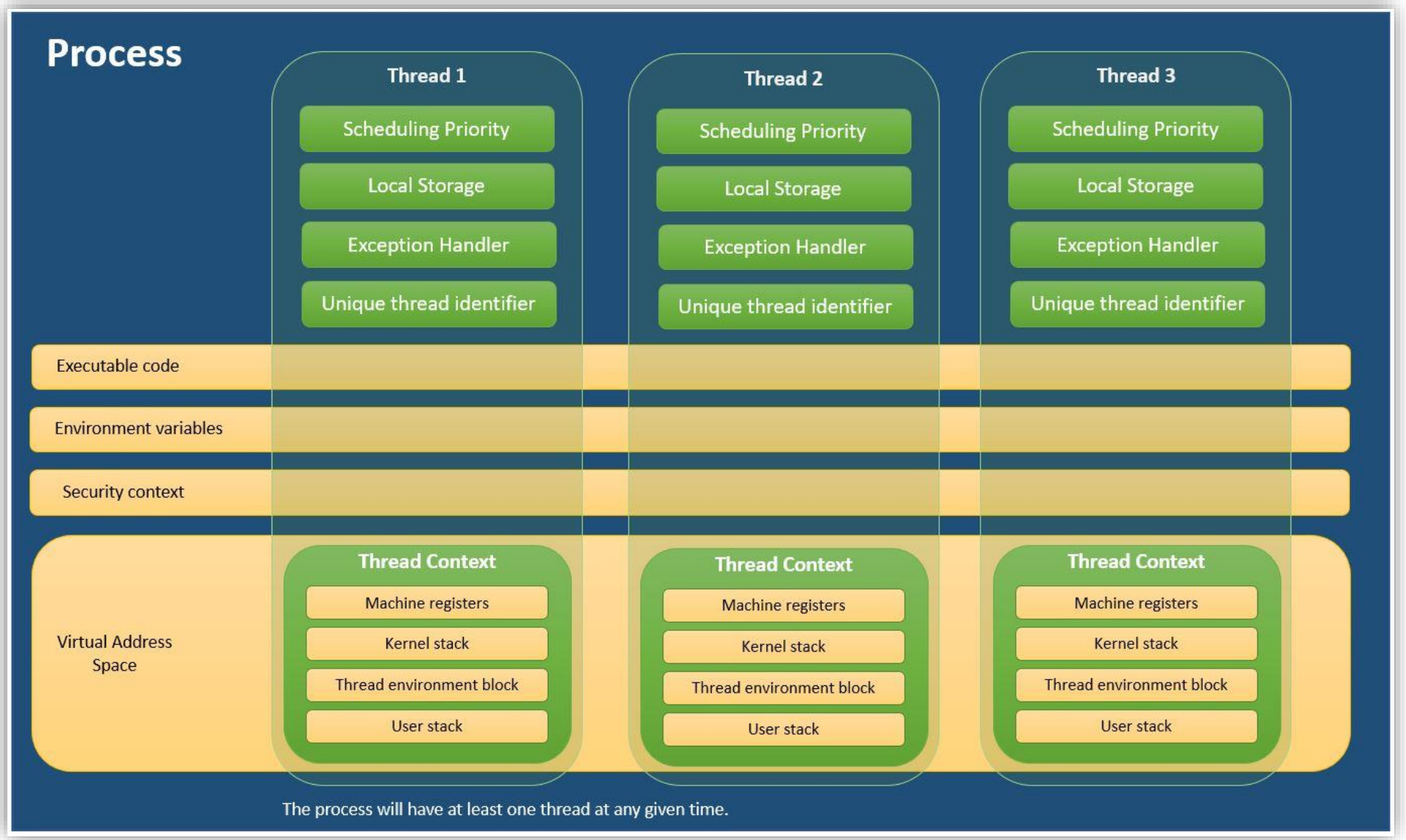
Multithreading & Multiprocessing

Multithreading	Multiprocessing/Multitasking
Multithreading is the ability of an operating system to execute the different parts of the program, called threads, concurrently.	Multitasking is the ability of an operating system to accommodate more than one program in the memory at the same time.
In multithreading all threads share same process address space [memory].	Processes do not share the memory
Threads can share variables since threads are in same process.	External resources like file system or database must be used for communication between multiple processes and each one runs under its unique process address space.
Inter thread communication is simple	Inter process communication is complex
Less resources required to launch/run them since they are part of an already running process, hence they are called as light weight tasks.	CPU has to spend more resources for launching a new process hence process is known as heavy weight.

Example of some Multithreading Applications

- MS-Word – Auto Saving, Spell Checking, Printing while editing
- VS.NET – Intellisense, Auto Compilation.
- Browser – Downloading multiple files and browsing simultaneously.
- Media Player – Movie and Sound
- and so on ….
- ….almost all program is multithread

Process vs Thread



Main Thread and Child Tread

```
class Program
{
    //MAIN THREAD
    0 references
    static void Main(string[] args)
    {
        Thread thread =Thread.CurrentThread;
        thread.Name = "Console main";
        Console.WriteLine(thread.Name);
        Console.WriteLine(thread.IsAlive);
    }
}
```

```
static void Print2()
{
    for (int i = 0; i < 10; i++)
    {
        //artificial delay for test
        Thread.Sleep( millisecondsTimeout: 100);

        Console.WriteLine("Some message");
    }

}
```

```
static void Main(string[] args)
{
    string message1 = "First Message";
    string message2 = "Second Message";
    //From main thread (Will be execute synchronously)
    Print(message1); Print(message2);
    //from Child thread 1 (Will be execute asynchronously)
    Thread threadWithParameter1 = new Thread(Print);
    threadWithParameter1.Start(message1);
    //from Child thread 2 (Will be execute asynchronously)
    Thread threadWithParameter2 = new Thread(new ParameterizedThreadStart(Print));
    threadWithParameter2.Start(message2);
    //from Child thread 3 (Will be execute asynchronously)
    Thread threadWithoutParameter = new Thread(new ThreadStart(Print2));
    threadWithoutParameter.Start();
}
//Note that: In thread operations, the method parameter
//must be object to match delegate
4 references
static void Print(object message)
{
    for (int i = 0; i <10; i++)
    {
        //artificial delay for test
        Thread.Sleep( millisecondsTimeout: 100);

        Console.WriteLine(message);
    }

}
```

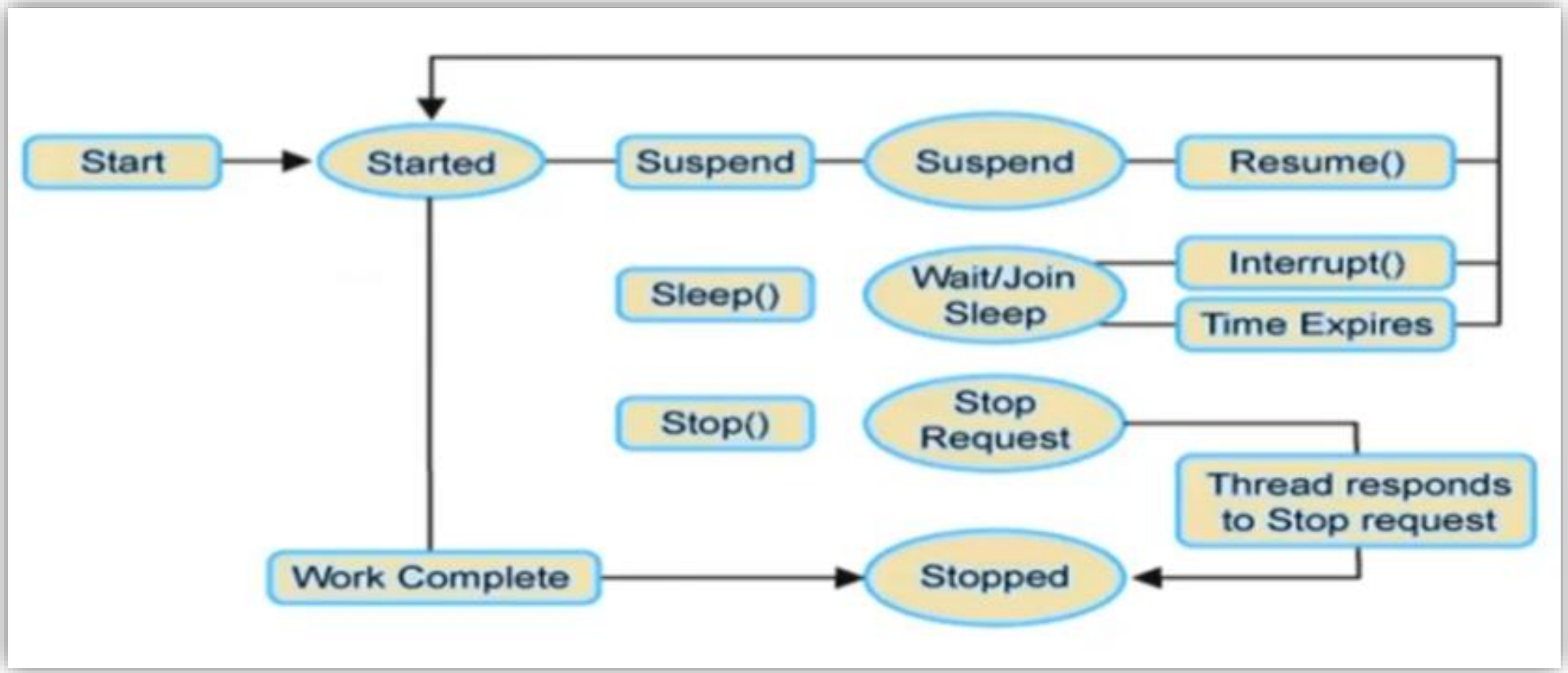
Important Notes

- The process terminates when the main thread terminates.
- A thread terminates only when the other **non-background** threads which it has created terminates.
- The creator doesn't wait for the **background** thread if it has to terminate.

Thread Operations

- **Start(), Start(params)** starts the thread
- **Thread**.Sleep(miliseconds) pauses tastk, that is in contained thread
- **Join():** If a thread executes t.Join() the current thread state is changed to waiting and it remains in that state until the thread referred by “t” is terminated.
- **Join(3000)**: The current threads waits for a max of 3000 milliseconds and then would resume automatically.
- **IsAlive()** : To check if the thread is dead or alive.
- **Thread.CurrentThread :** To Get the reference of the current thread.
- **t.ThreadPriority = ThreadPriority.Highest.**
- **Abort():** To stop the thread. When the Abort() method is called on a thread it throws **ThreadAbortException** irrespective of its state. If this exception is unhandled the thread terminates, and if it is handled and if **Thread.ResetAbort()** is executed the thread is not aborted and would continue normal execution.
- **IsBackground:** When a thread is created as background thread, the creator thread does not wait for the background thread to terminate or join the creator thread. The background thread is automatically aborted when the creator thread aborts.

Thread Lifecycle



Join( ) example

```
//from Child thread 1 (Will be execute asynchronously)
Thread threadWithParameter1 = new Thread(Print);

//from Child thread 2 (Will be execute asynchronously)
Thread threadWithParameter2 = new Thread(new ParameterizedThreadStart(Print));

//from Child thread 3 (Will be execute asynchronously)
Thread threadWithoutParameter = new Thread(new ThreadStart(Print2));


threadWithParameter1.Start(message1);
threadWithParameter1.Join( millisecondsTimeout: 5000);
//following will start after 5 sec

threadWithParameter2.Start(message2);
threadWithParameter2.Join( millisecondsTimeout: 3000);
//following will start after 3 sec

threadWithoutParameter.Start();//starts after 8 secs
threadWithoutParameter.Join( millisecondsTimeout: 0);
```

Thread States and properties

- IsAlive
- Priority
- IsBackgroud
- ThreadState
  - Running
  - Stopped
  - Aborted
  - Suspended
  - Unstarted

Thread Priority

```
//from Child thread 2 (Will be execute asynchronously)
Thread threadWithParameter2 = new Thread(new ParameterizedThreadStart(Print));

//from Child thread 3 (Will be execute asynchronously)
Thread threadWithoutParameter = new Thread(new ThreadStart(Print2));


//scheduler of OS give priority to threads when allocation
threadWithParameter1.Priority = ThreadPriority.Lowest;
threadWithParameter2.Priority = ThreadPriority.Lowest;
threadWithoutParameter.Priority = ThreadPriority.Highest;

threadWithParameter1.Start(message1);
threadWithParameter2.Start(message2);
threadWithoutParameter.Start();
```

Thread Pool

- A *thread pool* is a collection of threads that can be used to perform a number of tasks in the background.
- Thread pools are often employed in server applications.
- Each incoming request is assigned to a thread from the thread pool, so the request can be processed asynchronously, without tying up the primary thread or delaying the processing of subsequent requests.
- Once a thread in the pool completes its task, it is returned to a queue of waiting threads, where it can be reused.
- This reuse enables applications to avoid the cost of creating a new thread for each task
- Thread pools typically have a maximum number of threads.
- If all the threads are busy, additional tasks are placed in queue until they can be serviced as threads become available.

Application of Thread Pool

```
1 reference
public static void Run(object state)
{
    Console.WriteLine(Thread.CurrentThread.GetHashCode() + " : " + state);
    Thread.Sleep( millisecondsTimeout: 1000);
}

0 references
static void Main(string[] args)
{
    ThreadPool.SetMinThreads( workerThreads: 2,  completionPortThreads: 2);

    ThreadPool.SetMaxThreads( workerThreads: 4,  completionPortThreads: 4);

    WaitCallback wcb = new WaitCallback(Run);

    for (int i = 0; i < 10; i++)
    {
        ThreadPool.QueueUserWorkItem(wcb, i);

    }

    Thread.Sleep( millisecondsTimeout: 10000);
}
```

Foreground and Background Threads

- **Foreground Thread**
  - Foreground threads are those threads that keep running even after the application exits or quits.
  - It has the ability to prevent the current application from terminating.
  - The CLR will not shut down the application until all Foreground Threads have stopped.
- **Background Thread**
  - Background Threads are those threads that will quit if our main application quits. In short, if our main application quits, the background thread will also quit.
  - Background threads are views by the CLR and if all foreground threads have terminated, any and all background threads are automatically stopped when the application quits.

```
static void Main(string[] args)
{
    //Foreground thread
    //Thread threadForeground =new Thread(CreateFile);
    //threadForeground.Start();
    //Console.WriteLine("Files have been created");

    //Background thread
    Thread threadBackground = new Thread(CreateFile);
    threadBackground.IsBackground = true;
    threadBackground.Start();
    //Thread.Sleep(3000); //if not wont be executed
    //or to wait use readline this will keep main thread alive
    Console.ReadLine();
    Console.WriteLine("Files have been created");
}
1 reference
static void CreateFile()
{
    for (int i = 0; i < 20; i++)
    {
        Thread.Sleep(millisecondsTimeout:1000);
        var path = $"d:\\{DateAndTime.Now.Ticks}.txt";
        System.IO.File.WriteAllText(path, contents:"Data: " + i);
    }
}
```

Thread Synchronization, Thread safe classes and operations

- **Requirement:** In situation where one object is shared by more than one thread, if the thread has to modify the state of the object it should ensure that only one thread does so at a given instance of time.
- **Critical Section** is a block of code which can be executed by only one thread at any given instance of time.
- If there are two threads executing on same shared object then both the threads can execute some method on the object at the same point of time and if they then change the state of the object, it may result in ambiguity of data causing **Threads De-Synchronization.** To avoid this we use **lock block.**

```
//Monitoring object by lock keyword In C#
lock(object)
{
      //……. modify object here
}
```

Monitoring object by lock keyword in C#

- ***Managed by single process, does not affects between different processes
- Monitor(lock) is a .Net specific object and is local to a given process and thus it cannot be used for synchronizing threads running in different processes

```
static void Main(string[] args)
{
    Account account = new Account( owner: "Aydın",  balance: 1000);

    //Synchronous
    //account.WithDrawMoney(500);
    //account.DepositMoney(500);

    //Asynchronous
    for (int i = 0; i < 50; i++)
    {
        Thread threadWithDrawMoney = new Thread(account.WithDrawMoney);
        threadWithDrawMoney.Start( parameter: 500);
        Thread threadDepositMoney = new Thread(account.DepositMoney);
        threadDepositMoney.Start( parameter: 500);
    }

    Thread.Sleep( millisecondsTimeout: 5000);
    Console.WriteLine(account.Balance);
    Console.ReadLine();
}
```

```
class Account
{
    1 reference
    public Account(string owner, decimal balance)
    {
        Balance = balance; Owner = owner;
    }
    1 reference
    public string Owner { get; private set; }
    6 references
    public decimal Balance { get; private set; }

    1 reference
    public void WithDrawMoney(object amount)
    {
        lock (this)
        {
            Balance = Balance - Convert.ToDecimal(amount);
        }
    }
    1 reference
    public void DepositMoney(object amount)
    {
        lock (this)
        {
            Balance = Balance + Convert.ToDecimal(amount);
        }
    }
}
```

Mutex class for Thread

- ***Managed by Operating System, can be used one mutex between different processes
- It is a synchronization resource managed by the OS. Thus it can be used for synchronizing threads running in different processes.

```
private static Mutex mutex = new Mutex();
0 references
static void Main(string[] args)
{
    Account account = new Account( owner: "Aydın",  balance: 1000);
    for (int i = 0; i < 50; i++)
    {
        Thread threadWithDrawMoney = new Thread(account.WithDrawMoney);
        threadWithDrawMoney.Start( parameter: 500);
        Thread threadDepositMoney = new Thread(account.DepositMoney);
        threadDepositMoney.Start( parameter: 500);
    }
    Thread.Sleep( millisecondsTimeout: 5000);
    Console.WriteLine(account.Balance);
    Console.ReadLine();
}
```

```
class Account
{
    1 reference
    public Account(string owner, decimal balance)
    {
        Balance = balance; Owner = owner;
    }
    1 reference
    public string Owner { get; private set; }
    6 references
    public decimal Balance { get; private set; }
    1 reference
    public void WithDrawMoney(object amount)
    {
        mutex.WaitOne();
        Balance = Balance - Convert.ToDecimal(amount);
        mutex.ReleaseMutex();
    }
    1 reference
    public void DepositMoney(object amount)
    {
        mutex.WaitOne();
        Balance = Balance + Convert.ToDecimal(amount);
        mutex.ReleaseMutex();
    }
}
```

Semaphore class for Thread

- Semaphore is used for synchronizing threads.
- Within same process or in different process.
- Works exactly like Mutex but has a facility to allow more than one thread (but a pre-defined count) to execute a given block at the same time.
- Mutex can be treated as a special case of semaphore where count = 1.

```
class Demo1
{
    //static Semaphore s = new Semaphore(3,3);
    Semaphore s = new Semaphore(initialCount:1, maximumCount:5, name:"Test");
    1 reference
    public void Run()
    {
        Console.WriteLine(Thread.CurrentThread.Name + " Started");
        s.WaitOne();
        Thread.Sleep(millisecondsTimeout:1000);
        s.Release();
        Console.WriteLine(Thread.CurrentThread.Name + " Ended");
    }
}

0 references
class SemaphoreDemo
{
    0 references
    public static void Main()
    {
        for (int i = 0; i < 15; i++)
        {
            Demo1 d = new Demo1();
            Thread t = new Thread(new ThreadStart(d.Run));
            t.Name = "T" + i;
            t.Start();
        }
        Console.WriteLine("Main method ends");
    }
}
```

Task Parallel Library

- Types of TPL: Task Parallelizm and Data Parallelizm
- The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks.
- The purpose of the TPL is to make developers more productive by simplifying the process of adding parallelism and concurrency to applications.
- The TPL scales the degree of concurrency dynamically to most efficiently use all the processors that are available.
- In addition, the TPL handles the partitioning of the work, the scheduling of threads on the ThreadPool, cancellation support, state management, and other low-level details.
- By using TPL, you can maximize the performance of your code while focusing on the work that your program is designed to accomplish.
- Note: However, not all code is suitable for parallelization; for example, if a loop performs only a small amount of work on each iteration, or it doesn't run for many iterations, then the overhead of parallelization can cause the code to run more slowly.
- Parallelization like any multithreaded code adds complexity to your program execution.
- Data parallelism refers to scenarios in which the same operation is performed concurrently (that is, in parallel) on elements in a source collection or array. In data parallel operations, the source collection is partitioned so that multiple threads can operate on different segments concurrently.

System.Threading.Tasks.Parallel

- Parallel.For
- Parallel.ForEach
- Parallel.Invoke

```
static void Main(string[] args)
{
    Console.WriteLine("Main Starts");//sequential
    Parallel.Invoke( params actions: Work1,Work2);//parallel  only waits above
    Console.WriteLine("Main Ends");//sequential waits all above
}
1 reference
static void Work1()
{
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep( millisecondsTimeout: 100);
        Console.WriteLine($"Work1");
    }
}
1 reference
static void Work2()
{
    for (int i = 0; i < 10; i++)
    {
        Thread.Sleep( millisecondsTimeout: 100);
        Console.WriteLine($"Work2");
    }
}
```

```
static void Main(string[] args)
{
    Console.WriteLine("Main Starts");//sequential
                                     // Parallel.Invoke(Work1,Work2);//parallel  only waits above
    int m = 5;
    Parallel.For( fromInclusive: 1,  toExclusive: m,  body: (i :int ) =>
    {
        Console.WriteLine("Start: " +i);
        Work1(i);
        Console.WriteLine("Middle: " + i);
        Work2(i);
        Console.WriteLine("End: " + i);
    });
    Console.WriteLine("Main Ends");//sequential waits all above
}
1 reference
static void Work1(object i)
{
    Thread.Sleep( millisecondsTimeout: 1000);
    Console.WriteLine($"Work1: " + i);
}
1 reference
static void Work2(object i)
{
    Thread.Sleep( millisecondsTimeout: 1000);
    Console.WriteLine($"Work2: " + i);
}
```

Task Class

- Like thread executed parallel
- Task.WaitAny(,,,,)
- Task.WaitAll(,,,)
- Task.WhenAll(,,,)
- Like background thread
- Task task; task.Start(…);
- Task.Factory.StartNew(…) for direct create and start in one line.
- Task task; task.ContinueWith(…);
- :……..

```
static void Main(string[] args)
{
    Console.WriteLine("Main Starts");//sequential
    //EXECUTED AS PARALLEL
    Task task1 = new Task(() =>
    {
        Work1(Task.CurrentId);//SEQUENTIAL 1
        Work2(Task.CurrentId);//SEQUENTIAL 2
    });
    //EXECUTED AS PARALLEL
    Task task2 = new Task(() =>
    {
       Work1(Task.CurrentId);//SEQUENTIAL 1
       Work2(Task.CurrentId);//SEQUENTIAL 2
    });
    task1.Start();
    task2.Start();
    //task1.Wait();
    //task2.Wait();
    Task.WaitAll( params tasks: task1, task2); //Make above sequential for below
    Console.WriteLine("Main Ends");//sequential waits all above
}
2 references
static void Work1(object i)
{
    Thread.Sleep( millisecondsTimeout: 1000);
    Console.WriteLine($"Work1: " + i);
}
2 references
static void Work2(object i)
{
    Thread.Sleep( millisecondsTimeout: 0);
    Console.WriteLine($"Work2: " + i);
}
```

Concurrent Collections, Thread Safe Objets

- using System.Collections.Concurrent;
- ConcurrentDictionary<int, string> numbers = new ConcurrentDictionary<int, string>();
- ConcurrentStack
- ConcurrentQueue
- Etc……

Asynchronous Programming

- Using previous technique discussed in this lecture is hard to develop asynchronous programming.
- But, Asynchronous Programming has positive sides:
- Overcome performance issues
- Easy to debug
- Easy to write
- Easy to maintain
- Can be executed as synchronously by await keyword

async/await keywords

```
static async Task Main(string[] args)
{
    Console.WriteLine("Main Start");
    string result = await Method1();
    await Method2();
    Console.WriteLine("Main End");
    Console.ReadKey();
}
1 reference
static async Task<string> Method1()
{
    await Task.Run(() =>
    {
        for (int i = 1; i <= 10; i++)
        {
            Thread.Sleep(millisecondsTimeout:100);
            Console.WriteLine(" Method 1: " + i);
        }
    });

    return "Finished";
}
1 reference
static async Task Method2()
{
    await Task.Run(() =>
    {
        for (int i = 1; i <= 10; i++)
        {
            Thread.Sleep(millisecondsTimeout:100);
            Console.WriteLine(" Method 2: " + i);
        }
    });
}
```

Signaling in Asynchronous Programming

```
private static readonly AutoResetEvent _operationSignal = new AutoResetEvent(initialState:false);
0 references
static void Main(string[] args)
{
    SendMoney();//Waits another operation signal to continue
    SendSmsMessage();//take some time

    Thread.Sleep(millisecondsTimeout:1000);//wait for eof main
    Console.WriteLine("EOF Main");
}

1 reference
static Task SendMoney()
{
    return Task.Run(()=>
    {
        _operationSignal.Reset();//initialize
        Console.WriteLine("Op 1: Waiting Verification SMS Message for Transfer");//do some jobs here
        _operationSignal.WaitOne(); //wait signal.
        Console.WriteLine("Op 3: Successfully Sent Money");//will run after signal received
    });
}
1 reference
static Task SendSmsMessage()
{
    Thread.Sleep(millisecondsTimeout:2000);//take some time sms response

    return Task.Run(() =>
    {
        Console.WriteLine("Op 2: SMS Message Sent");
        _operationSignal.Set();//send finishing signal
    });
}
```

Cancellation Tokens in Async methods

```
public static void Main()
{
    var cancelSource = new CancellationTokenSource();
    new Thread(start: () =>
    {
        try
        {
            Work(cancelSource.Token); //).Start();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nCanceled!");
        }
    }).Start();

    Thread.Sleep(millisecondsTimeout: 1000);
    cancelSource.Cancel(); // Safely cancel worker.

}
1 reference
private static void Work(CancellationToken cancelToken)
{
    while (true)
    {
        Thread.Sleep(millisecondsTimeout: 10);
        Console.Write("345");
        cancelToken.ThrowIfCancellationRequested();
    }

}
```

```
public static void Main()
{
    var cancelSource = new CancellationTokenSource();
    var cancelToken = cancelSource.Token;
    new Thread(start: () =>
    {
        Work(cancelToken); //).Start();

    }).Start();

    Thread.Sleep(millisecondsTimeout: 1000);
    cancelSource.Cancel(); // Safely cancel worker.

}
1 reference
private static void Work(CancellationToken cancelToken)
{
    while (true)
    {
        Thread.Sleep(millisecondsTimeout: 20);
        if (cancelToken.IsCancellationRequested)
        {
            Console.WriteLine("\nCanceled!...");
            return;
        }
        Console.Write("345");
    }
}
```

Reflection in C#

Type Class

```
class Student
{
    //members must be public to read from reflection
    public int temp;
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public DateTime BirthDate { get; set; }
    0 references
    public void StudyLesson()
    {
        Console.WriteLine("...");
    }
}
```

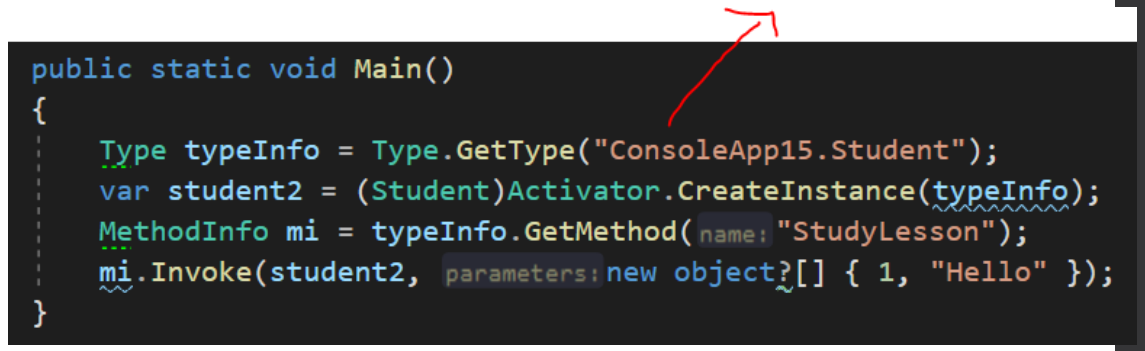
```
public static void Main()
{

    Type info = typeof(Student);

    FieldInfo[] fieldInfos = info.GetFields();
    foreach (var data :FieldInfo in fieldInfos)
        Console.WriteLine(data.Name);

    PropertyInfo[] propertyInfos = info.GetProperties();
    foreach (var data :PropertyInfo in propertyInfos)
        Console.WriteLine(data.PropertyType);

    MethodInfo[] methodInfos = info.GetMethods();
    foreach (var data :MethodInfo in methodInfos)
        Console.WriteLine(data.Name);
}
```

Activator Class

```
public static void Main()
{
    Type typeInfo = typeof(Student);
    var student1 = Activator.CreateInstance<Student>();
    //or
    var student2 = (Student) Activator.CreateInstance(typeInfo);

    MethodInfo mi = typeInfo.GetMethod( name: "StudyLesson");
    mi.Invoke(student2, parameters: new object?[] {1, "Hello"});
}
```

```
class Student
{
    //members must be public to read from reflection
    public int temp;
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public DateTime BirthDate { get; set; }

    0 references
    public void StudyLesson(int number, string message)
    {
        Console.WriteLine(message);
    }
}
```

From full qualified assembly name



Predefined Attributes

- Obsolete Attribute
- Conditional Attribute

```
[Obsolete]
0 references
class Student
{
    public int temp;
    0 references
    public int Id { get; set; }
    0 references
    public string Name { get; set; }
    0 references
    public DateTime BirthDate { get; set; }

    [Obsolete( message: "Future version of program will not use this class")]
    0 references
    public void StudyLesson(int number, string message)
    {
        Console.WriteLine(message);
    }
}
```

```
#define Test
using System;
using System.Diagnostics;
using System.Reflection;
namespace ConsoleApp15
{
    0 references
    class Program
    {
        0 references
        public static void Main()
        {
            Method();
        }
        [Conditional( conditionString: "Test")]
        1 reference
        static void Method()
        {
            Console.WriteLine("Conditional worked");
        }
    }
```

Custom Attributes

```
class MyAttribute : Attribute
{
    2 references
    public string Developer { get; set; }
    1 reference
    public int Version { get; set; }
}

[MyAttribute(Developer = "Aydin Secer",Version = 1)]
[Obsolete]
1 reference
class Program
{
    0 references
    public static void Main()
    {
        var typeInfo = typeof(Program);
        var attributes :IEnumerable<Attribute> = typeInfo.GetCustomAttributes();
        var attribute = typeInfo.GetCustomAttribute<MyAttribute>();
        Console.WriteLine(attribute.Developer);
    }
}
```

Garbage Collection

Data Security

New features in C#

Debugging Applications, Debugging Tools Call Stack Window, Locals and Autos Windows

LINQ (Language Integrated Query)

Some Revisited Topics

- Nullable Types
- Dynamic vs Static Keywords
- Reference vs Value Types
- Stack and Heap
- Compile Time vs. Runtime Errors

Working with XML

ADO.Net

Entity Framework

Refactoring C# Code

Defensive Coding

Design Patterns

Dependency Injection

Unsafe code and pointers

Working with Culture & Regions

Regular Expressions (Regex)

String, StringBuilder in C#

Association between Classes