# The Problem with Arrays

- **Fixed Size**: Cannot grow or shrink dynamically.
- **Homogeneous**: Stores only one type (or `object` with type safety issues).
- **Limited Functionality**: No built-in add/remove/search methods.
- **Solution**: Collections!

# What are Collections?

- Classes designed to store, manage, and manipulate groups of objects.
- Provide more flexibility and functionality than raw arrays.
- Found in `System.Collections` and `System.Collections.Generic` namespaces.

# Collections and Interface

- C# collections typically implement certain key interfaces which define their behavior:
  - **IEnumerable**: Provides the ability to **iterate** through the collection.
    - Readonly Secnario
  - **ICollection**: Defines size, enumerators, and **adding** and **removing** methods for all collections.
    - Manipulation Secnario
  - **IList**: Represents a collection of objects that can be individually accessed by index (inserting , removing).
    - Advanced List Operation
  - **IDictionary<TKey, TValue>**: Represents a collection of key-value pairs.

# Non-Generic Collections ( `System.Collections` )

- **Examples**:`ArrayList`, `Hashtable`.
- **Store**: Elements of type **`object`**.
- **Disadvantages**:
  - **Type Safety Issues**: No compile-time checking, runtime errors.
  - **Performance Overhead**: Boxing/Unboxing for value types.
  - **Recommendation**: Avoid in modern C# (unless legacy code).

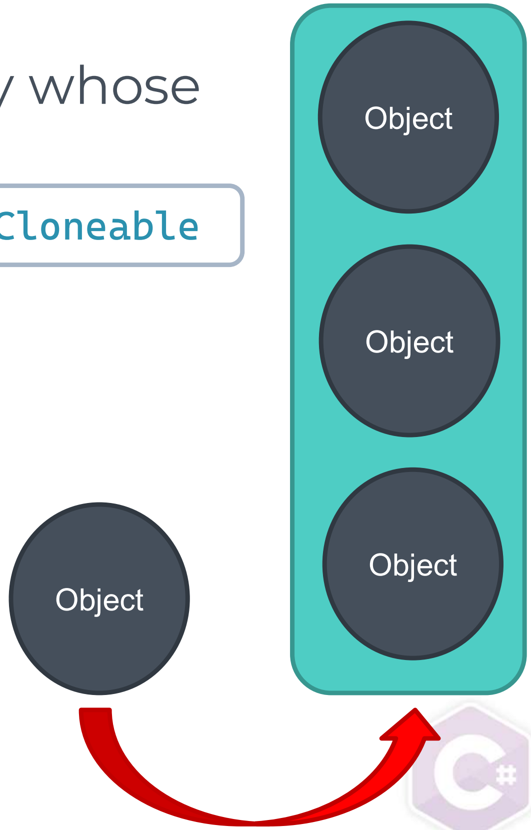
# ArrayList Collection

- Implements the **IList** interface using an array whose size is dynamically increased as required.

```
public class ArrayList : ICollection, IEnumerable, IList,ICloneable
```

- Methods
  - Add(Object)
  - Insert(Index,Object)
  - Remove(Object)
  - RemoveAt(index)
  - RemoveRange(start index, end index)
  - Clear()

```
ArrayList arlist = new ArrayList();
arlist.Add(10);
```

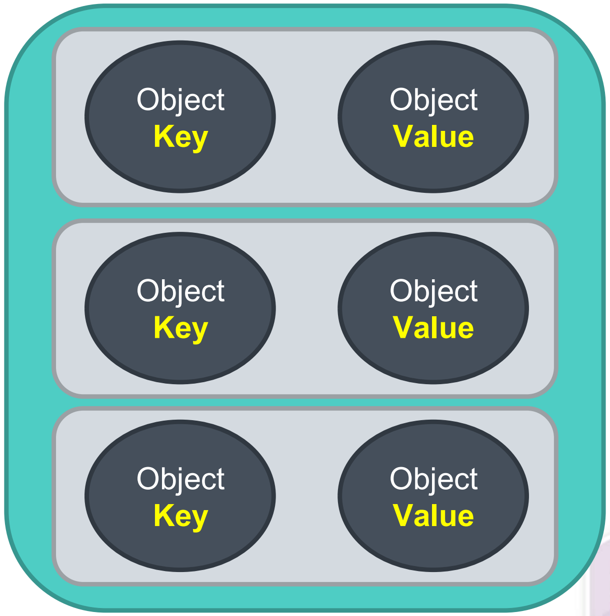


# ArrayList Collection

- Methods
  - `TrimToSize( )`
  - `Sort()`
  - `Reverse()`
  - `Object[] ToArray()`
  - `int indexOf(Object)`
  - `Contains(Object)` ➔ `Object.Equals()`
  - `[int index] indexer`
- Properties
  - `Capacity`
  - `Count`

# Hashtable Collection

- Store Data in Key-value format, where keys are unique and used in indexer
  - Ex: Dictionary ( word – meaning )
- Methods
  - `void Add(object key, object value)`
  - `void Clear()`
  - `bool ContainsKey(object key)`
  - `bool ContainsValue(object value)`
  - `void Remove(object key);`



# Hashtable Collection

- Properties
  - Count
  - Item[Key]
  - Keys
  - values

```
Hashtable ht = new Hashtable();
ht.Add("One", 1);
ht.Add("Two", 2);
ht.Add("three", 3);
Console.WriteLine(ht["three"].ToString()); // print 3
```

```
foreach (DictionaryEntry node in ht)
{
    Console.WriteLine(node.ToString()); // print 3
}
```

```
foreach (var k in ht.Keys)
{
    Console.WriteLine(k.ToString());
}
```

# The Need for Generics

- **Problem**: Non-generic collections lack type safety and performance.
- **Solution** Generics!
- **Generics**: Allow defining classes/methods with placeholders for types.
- Type is specified when used (e.g., `List<string>`).

## Benefits of Generics

- **Type Safety**: Enforced at compile time, fewer runtime errors.
- **Performance**: No boxing/unboxing for value types.
- **Code Reusability**: Write code once, use with multiple types.

# Generic Method

```
static void Swap (ref int x, ref int y)
{
        int temp;
        temp = x;
        x = y;
        y = temp;
}
```

```
static void Swap (ref char x, ref char y)
{
        char temp;
        temp = x;
        x = y;
        y = temp;
}
```

# Generic Method

- Definition

```
static void Swap<T>(ref T x, ref T y)
 {
     T temp;
     temp = x;
     x = y;
     y = temp;
 }
```

- Calling

```
Swap <char> (ref x, ref y);
```

```
Swap <int> (ref x, ref y);
```

```
Swap (ref x, ref y);
```

# Default generic value

- `default(T)`
  - Ex : return of pop Method

```
public int pop()
{
  if (tos > 0)
  {
   tos--;
   return stk[tos];
  }
  else
    return -1;
}
```

```
public T pop()
{
  if (tos > 0)
  {
    tos--;
    return stk[tos];
  }
  else
    return default(T);
}
```

# Generic Class

- Definition

Generic type

```
public class Demo <T>
{
        public T  v;
        public Demo(T x )
        { v=x;}
}
```

```
public class Pair <T,U>
{
        public T  v1;
        public U  v2;
        public Pair(T x,U y )
        { v1=x; v2=y; }
}
```

# Generic Class

- Declare Reference and Instantiating  an Object

Constructed type

```
Demo<int> D = new Demo<int>(10);
```

```
Pair<int, string> D2 = new Pair<int, string>(10,"Hi");
```

# Generic Interface

- Definition

```
public interface IGenInteface <T>
{
  T Prperty { get; set; }
}
```

# Generic Constraint

- Arithmetic operation Constraint

```
class Complex<T> where T : INumber<T>
{
    public T real;
    public T img;
    public Complex()
    {
        real = img = default;
    }
    public static Complex<T> operator +(Complex<T> c1, Complex<T> c2)
    {
        Complex<T> total = new Complex<T>();
        total.real = c1.real + c2.real; // Error cant apply operator + for T and T
    }
}
```

# Generic Type Constraint

- Constraint on `T` could be achieve using `where` statement

```
GenericTypeName<T> where T  : contraint1, constraint2
```

```
class GenericClass<T, U> where T : class1, Interface1
                         where U : new()
    { … }
```

## Generic Type Constraint

| | |
|---|---|
| class | The type argument must be any class, interface, delegate, or array type. |
| class? | The type argument must be a nullable or non-nullable class, interface, delegate, or array type. |
| struct | The type argument must be non-nullable value types such as primitive data types int, char, bool, float, etc. |
| new() | The type argument must be a reference type which has a public parameterless constructor. It cannot be combined with struct and unmanaged constraints. |
| notnull | Available C# 8.0 onwards. The type argument can be non-nullable reference types or value types. If not, then the compiler generates a warning instead of an error. |
| unmanaged | The type argument must be non-nullable unmanged types. |

# Generic Type Constraint

| | |
|---|---|
| base class name | The type argument must be or derive from the specified base class. The Object, Array, ValueType classes are disallowed as a base class constraint. The Enum, Delegate, MulticastDelegate are disallowed as base class constraint before C# 7.3. |
| <base  class name>? | The type argument must be or derive from the specified nullable or non-nullable base class |
| <interface name> | The type argument must be or implement the specified interface. |
| <interface name>? | The type argument must be or implement the specified interface. It may be a nullable reference type, a non-nullable reference type, or a value type |
| where T: U | The type argument supplied for T must be or derive from the argument supplied for U. |

## Generic Type Constraint

| | |
|---|---|
| INumber<T> | The type argument must be numeric type |
| IBinaryInteger<T> | The type argument must be integer |

# Generic and Inheritance

- Inheriting generic types

```
public class GenStack <T>
{
        public T  [ ] stk;
        public int  size;
}
```

```
class specialStack <T>:Genstack<T>
{
   …
}
```

```
class specialStack:Genstack<int>
{
   …
}
```

# Generic and Inheritance

- Implementing Generic Interface

```
public interface IGenInteface <T>
{
  T Prperty { get; set; }
}
```

```
public class GenClass2<T>:
IGenInteface<T>
    {
        T t1;
        public T Prperty
        {
            get
            {
                return t1;
            }
            set
            {
                t1 = value;
            }
        }
    }
```

```
 public class Class3 :
IGenInteface<int>
    {
        int t2;
        public int Prperty
        {
            get
            {
                return t2;
            }
            set
            {
                t2 = value;
            }
        }
    }
```

# Common Generic Collections

- `List<T>`
- `Dictionary<TKey, TValue>`
- `HashSet<T>`
- `Queue<T>`
- `Stack<T>`

# List<T> : Dynamic Array

- **Concept**: Resizable array, ordered collection.
- **Features**: Add(), Remove(), Insert(), Contains(), Sort().
- **Example**: List<string> names = new List<string>();
- **Demo**: Basic operations (add, remove, iterate).

```
List<int> l = new List<int>();
List<employee> empl = new List<employee>();
```

# `Dictionary<TKey, TValue>` : Key-Value Pairs

- **Concept**: Stores unique keys mapped to values.
- **Features**: Fast lookups by key.
- **Example**:
- `Dictionary<int, string> employees = new Dictionary<int, string>();`
- **Demo**: Add, retrieve, update, remove by key.

```
var Numbers2 =
new Dictionary<int, string>
    {
        {19, "nineteen" },
        {23, "twenty-three" },
        {42, "forty-two" }
    };
```

```
 var numbers =
new Dictionary<int, string>
  {
      [7] = "seven",
      [9] = "nine",
      [13] = "thirteen"
  };
```

## HashSet<T> : Unique Elements

- **Concept**: Stores a collection of unique elements.
- **Features**: Optimized for fast membership testing (`Contains()`).
- **Does not maintain order**.
- **Example**:

```
HashSet<string> uniqueWords = new HashSet<string>();
```

## `Queue<T>`: First-In, First-Out (FIFO)

- **Concept**: Elements added to one end, removed from the other.
- **Methods**: `Enqueue()` (add), `Dequeue()` (remove), `Peek()` (view next).
- **Example**:

```
Queue<string> tasks = new Queue<string>();
```

## Stack<T>: Last-In, First-Out (LIFO)

- **Concept**: Elements added and removed from the same end.
- **Methods**: `Push()` (add), **`Pop()`** (remove), **`Peek()`** (view top).
- **Example**:

```
Stack<int> history = new Stack<int>();
```