Programming in C#. Fundamentals

SIGMA Software

Lesson 4
Collections and Generics

Collections and Generics

Arrays
Generic Lists
LINQ

Arrays

An array is an object (not just a stream of objects). See System.Array.

Bounds checking is performed for all access attempts.

Declaration similar to Java, but more strict.

- Type definition: *a* is a “1D array of int’s”
- Instance specifics: *a* is equal to a 1D array of int’s of size 10.

```
int[] a = new int[10];
int[] b = a;
int[] c = new int[3] { 1, 2, 3 };
int[] d = { 1, 2, 3, 4 };
```

Jagged arrays

Can have standard C-style *jagged* arrays

- Stored in random parts of the heap
- Stored in row major order

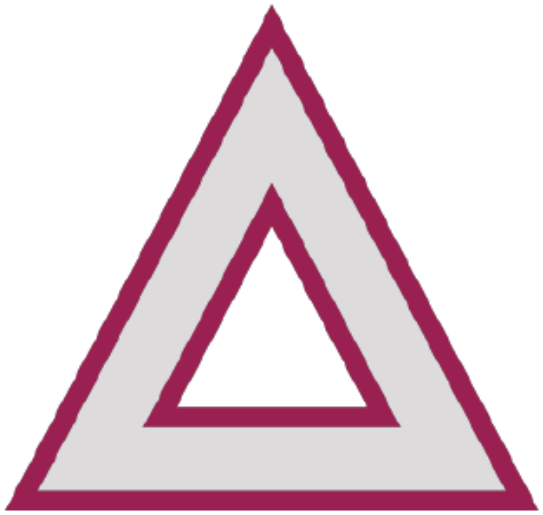
```
int[] array = new int[30];
int[][] array = new int[2][];
array[0] = new int[100];
array[1] = new int[1];
```

Multi-dimensional arrays

- Multi-dimensional arrays are jagged arrays with a *user-enforced* constraint in C.
  - Really just 1D arrays, but each element can contain a 1D array (recursion).
- C# provides ***true*** multi-dimensional arrays
  - Elements are stored sequentially
  - CLR (JIT compiler) computes the offset code

```
int[,] array = new int[10, 30];
array[3, 7] = 137;
int[,] arr4 = new int[2, 3] { { 1, 2, 3 }, { 4, 5, 6 } };
int[,] arr5 = new int[,] { { 1, 2, 3 }, { 4, 5, 6 } };
int[,] arr6 = { { 1, 2, 3 }, { 4, 5, 6 } };
```

Limitations of Arrays



Must declare size:

- Too large: waste space
- Too small, run out of room

Easy to run past the end (crash)

- Person oops = peopleArray[7];

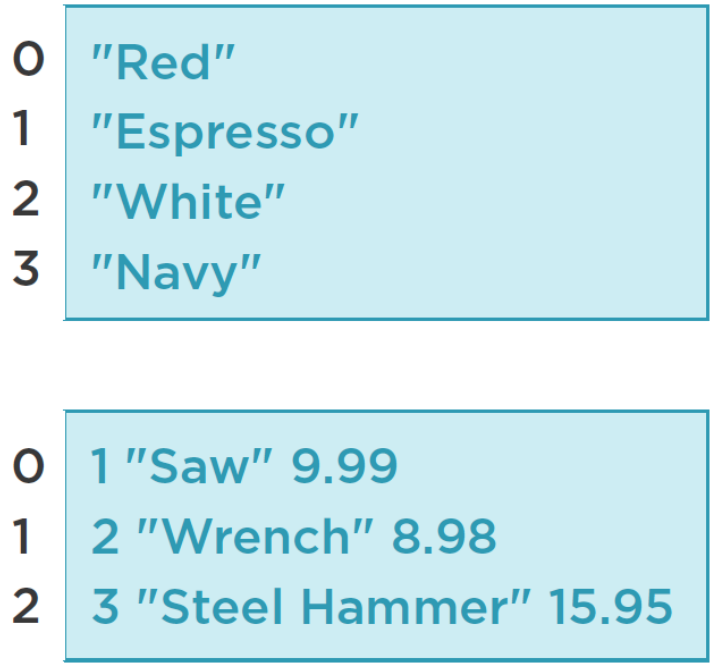
Foreach Loop

```
peopleArray[0] = new Person() { Name = "John" };
peopleArray[1] = new Person() { Name = "Paul" };
peopleArray[2] = new Person() { Name = "George" };
peopleArray[3] = new Person() { Name = "Ringo" };
peopleArray[4] = new Person() { Name = "Frodo" };
peopleArray[5] = new Person() { Name = "Merry" };
peopleArray[6] = new Person() { Name = "Pippin" };

foreach (Person person in peopleArray) {
    Console.WriteLine($"Name = {person.Name}");
}
```

Generic Lists

A strongly typed list of elements that is accessed using a positional index number



Array vs. Generic List

Array	Generic List
Strongly typed	Strongly typed
Fixed length	Expandable
No ability to add or remove elements	Can add, insert, or remove elements
Multi-dimensional	One-dimensional

LIST<T>

List<T>

List<int>

List<decimal>

List<string>

List<Product>

LIST<T>

- Declaring and Populating a Generic List
- Using Collection Initializers
- Initializing a List of Objects
- Retrieving an Element from a Generic List
- Iterating Through a Generic List
- Types of C# Lists
- FAQ

Declaring and Initializing a Generic List

```
List<string> colorOptions;
```

- List of what?
- List<T>
  - Where T is the type of elements the list contains

"Red"
"Espresso"
"White"
"Navy"

```
List<string> colorOptions;
colorOptions = new List<string>();
```

```
List<string> colorOptions = new List<string>();
```

```
var colorOptions = new List<string>();
```

Populating a List (Add)

```
colorOptions.Add("Red");
colorOptions.Add("Espresso");
colorOptions.Add("White");
colorOptions.Add("Navy");
```

"Red"
"Espresso"
"White"
"Navy"

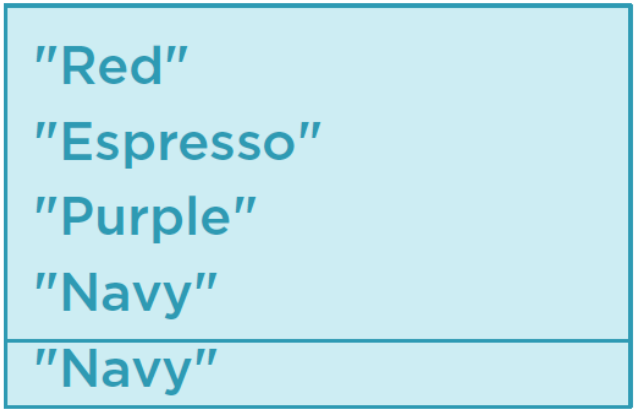
Populating a List (Insert)

```
colorOptions.Insert(2, "Purple");
```

"Red"
"Espresso"
"Purple"
"White"
"Navy"

Removing an Element

```
colorOptions.Remove("White");
```



Generic List Best Practices

Do:

Use generic lists to manage collections

Use Add over Insert where possible

Use a plural variable name for the list

Avoid:

Removing elements where possible

Collection Initializers

```
var colorOptions = new List<string>();
```

```
colorOptions.Add("Red");
colorOptions.Add("Espresso");
colorOptions.Add("White");
colorOptions.Add("Navy");
```

```
var colorOptions = new List<string>() {"Red","Espresso","White","Navy"};
```

Iterating a List

foreach	for
Quick and easy	Complex but flexible
Iterate all elements	Iterate all or a subset of elements
Element is read-only But the element's properties are editable	Element is read/write

Common C# Lists by Namespace

System

- Array

System.Collections (.NET 1)

- ArrayList

System.Collections.Generic (.NET 2+)

- List<T>
- LinkedList<T>
- Queue<T>
- Stack<T>

Generic Dictionaries

A strongly typed collection of keys and values

Key:

Must be unique

Must not be changed

Cannot be null

List vs. Dictionary

List	Dictionary
Contains elements	Contains elements defined as key and value pairs
Accessed by a positional index	Accessed by key
Allows duplicate elements	Allows duplicate values but unique keys
Marginally faster iteration	Marginally faster look ups

Generic Dictionary

Dictionary<TKey, TValue>

Dictionary<int, int>

Dictionary<int, string>

Dictionary<string, string>

Dictionary<int, Product>

Dictionary<string, Product>

Declaring a Generic Dictionary

```
Dictionary<string, string> states;
```

- Dictionary of what?
  - Value
  - Key
- Dictionary<TKey, TValue>
  - TKey is the type of the key
  - TValue is the type of the value

"CA"	"California"
"WA"	"Washington"
"NY"	"New York"

```
Dictionary<string, string> states;
states = new Dictionary<string, string>();
```

```
Dictionary<string, string> states =
                          new Dictionary<string, string>();
```

```
var states = new Dictionary<string, string>();
```

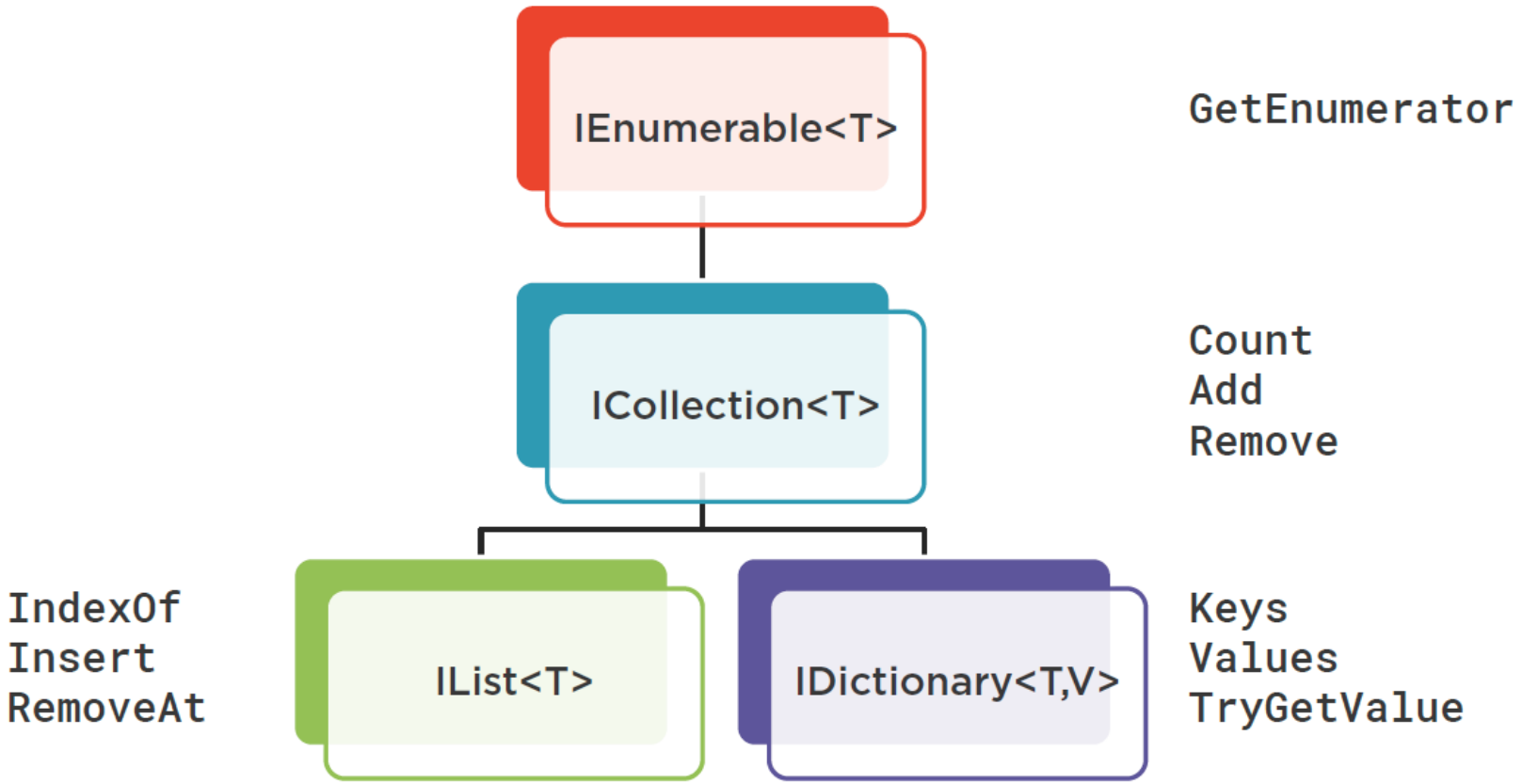
Declaring and Populating a Dictionary

```
var states = new Dictionary<string, string>();
```

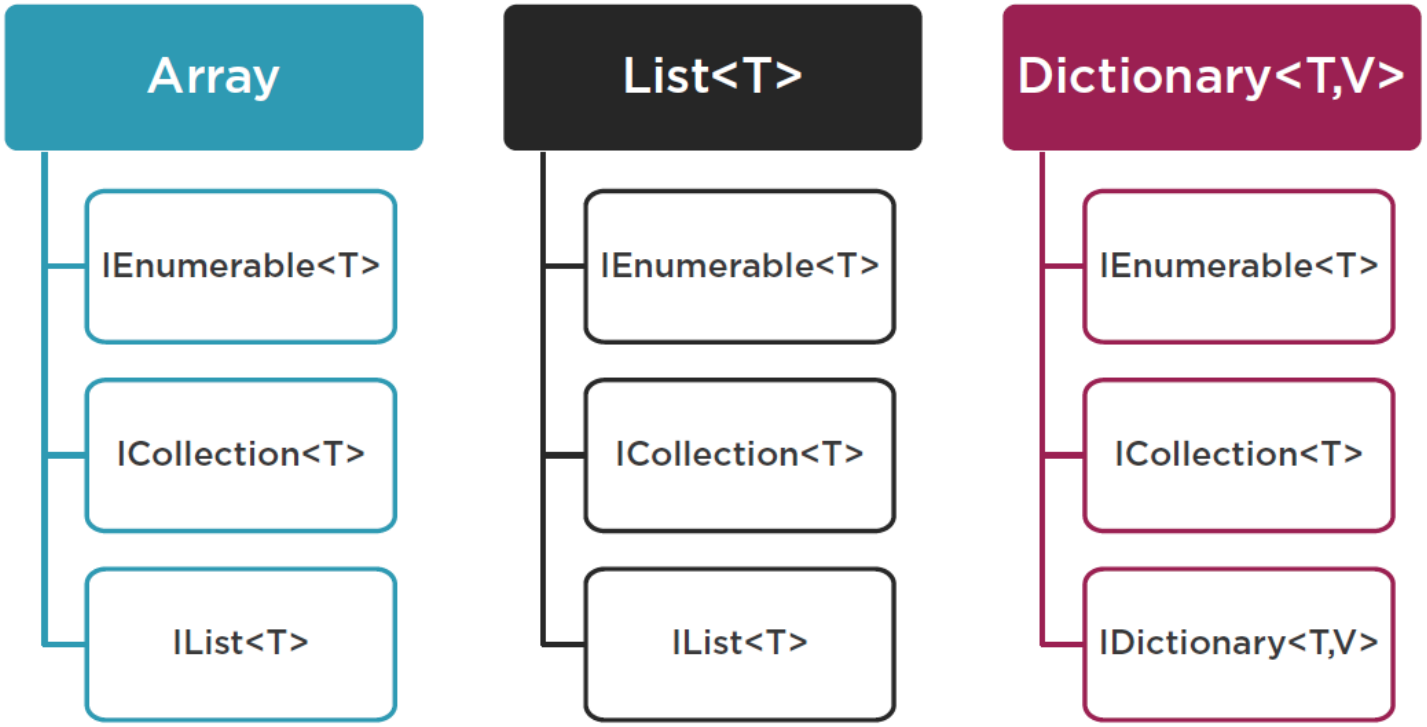
```
states.Add("CA", "California");
states.Add("WA", "Washington");
states.Add("NY", "New York");
```

```
var states = new Dictionary<string, string>()
{
    {"CA", "California" },
    {"WA", "Washington"},
    {"NY", "New York" },
};
```

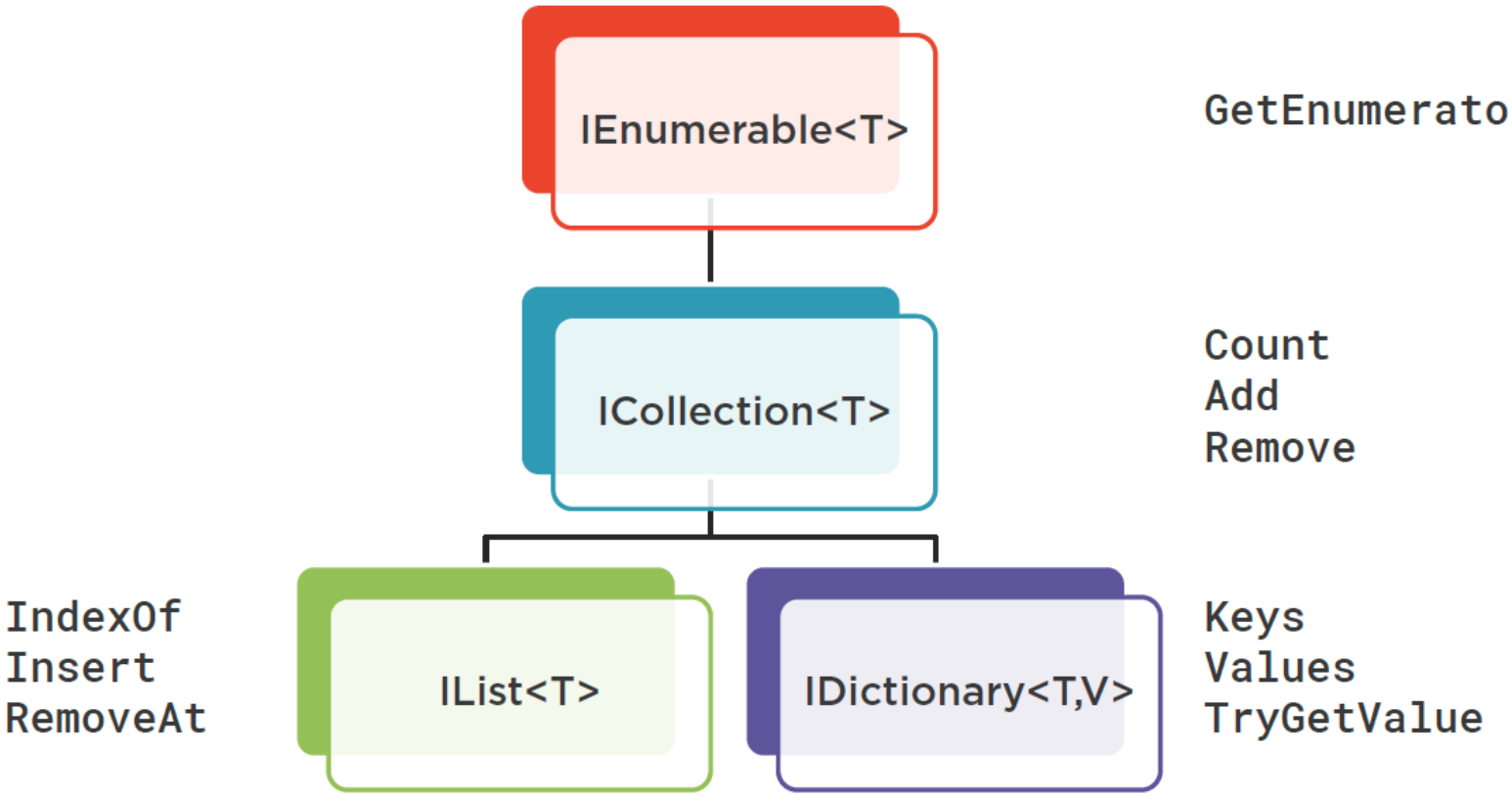
Generic Collection Interfaces



Generic Collection Interfaces



IEnumerable<T>



Returning IEnumerable<T>

Best Practices

Do:

Consider returning an IEnumerable<T> to provide an immutable collection

Consider returning IEnumerable<T> when the calling code use cases are unknown

Consider returning IQueryable<T> when working with a query provider, such as LINQ to SQL or Entity Framework

Avoid:

Returning an IEnumerable<T> if the collection must be modified by the caller

Returning an IEnumerable<T> if the caller requires information about the collection, such as the count

Returning an IEnumerable<T> if the caller must be notified of a change to the collection

Returning a Collection from a Method

IEnumerable<T>	IList<T> or ICollection<T>	List<T>, Array[] or Dictionary<T,V>
Read-only sequence of elements	Flexible updatable collection	Specific updateable collection

LINQ

Language INtegratedQuery

A way to express queries against a data source directly from a .NET language, such as C#.

Query Syntax

```
var vendorQuery = from v in vendors
                  where v.CompanyName.Contains("Toy")
                  orderby v.CompanyName
                  select v;
```

Method Syntax

```
var vendorQuery = vendors
                .Where(v => v.CompanyName.Contains("Toy"))
                .OrderBy(v=> v.CompanyName);
```

Extension Method

A method added to an existing type without modifying the original type.

Building a LINQ Query: Query Syntax

```
var vendorQuery = from v in vendors
                  where v.CompanyName.Contains("Toy")
                  orderby v.CompanyName
                  select v;
```

Building a LINQ Query: Method Syntax

```
var vendorQuery = vendors.Where(FilterCompanies)
                         .OrderBy(OrderCompaniesByName);


private bool FilterCompanies(Vendor v) =>
                            v.CompanyName.Contains("Toy");

private string OrderCompaniesByName(Vendor v) =>
                            v.CompanyName;
```

Using Lambda Expressions

```
var vendorQuery = vendors
                  .Where(v => v.CompanyName.Contains("Toy"))
                  .OrderBy(v => v.CompanyName);
```

LINQ and Collections

```
vendors.Where(v =>
        v.CompanyName.Contains("Toy")

vendors.OrderBy(v => v.CompanyName)

vendors.Select(v=>v.Email);

vendors.GroupBy(v => v.Category);

vendors.Average(v => v.NoOfProducts);

vendors.First(v => v.VendorId == 22);

vendors.FirstOrDefault(v =>
                  v.VendorId == 22);

vendors.Where(v =>
        v.CompanyName.Contains("Toy"))
  .GroupBy(v => v.Category)
```

- Filter based on criteria
- Sort on defined string
- Shape by selecting properties
- Group by a defined property
- Aggregate elements
- Return the first matching element
- Return the first matching element or null
- Combine as needed

Q & A

Practice
Lesson 4

Home work