Collections and Generics in C# 10

Introducing Collections and Generics

Zoran Horvat
CEO at Coding Helmet

@zoranh75 https://codinghelmet.com

In This Module...

Outline division of collections

Learn *why* do we need collections

Collections and Generics in C# 10

Version Check

Version Check

This version was created by using:

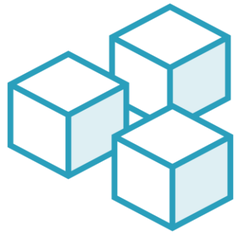
- .NET 6
- C# 10
- Visual Studio Code 1.63

Relevant Notes

A note on frameworks and libraries:

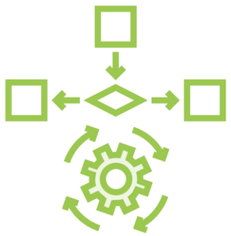
- The `BenchmarkDotNet` library used in this course is v0.13.1
- Future versions might include breaking changes
- Refer to https://benchmarkdotnet.org/ for latest information

Assumptions About Previous Experience



This course does not teach collections fundamentals

Do you know how `List<T>` stores items?



Nor does it teach basics of generics

What does `<T>` mean?

Learning collections fits before learning LINQ

There will only be a couple of most trivial calls to LINQ

```
IEnumerable<string> seq = new[]
{
    "something", "again", "and again"
};

string first = seq.First();
               // "something"

string single = seq.Single(); // throws


IEnumerable<string> head = seq.Take(2);
                  // "something", "again"

IEnumerable<string> tail = seq.Skip(1);
                  // "again", "and again"
```

- LINQ defines extension methods on `IEnumerable<T>`
- Applies to collection or sequence of type T

- Throws if sequence is empty

- Throws if sequence is empty, or contains more than one element

- Takes at most two items

- Skips one item and returns the rest

```
IEnumerable<Money> GetHourlyRates(IEnumerable<Worker> workers)
{
    foreach (Worker worker in workers)
    {
        yield return worker.HourlyRate;
    }
}
```

This course assumes good understanding of sequences

`IEnumerable<T>` **and** `yield return` **will be used in demos**

Virtually all collections depend on `IEnumerable<T>`

```
public class SomeClass
{
    private List<string> Data { get; }
    ...
    public List<string> GetData() => this.Data;
}
```

This course assumes fair understanding of object-oriented design

What is wrong with method above?

```
public class SomeClass
{
    private List<string> Data { get; }
    ...
    public List<string> GetData() => this.Data;
}

SomeClass someObject = new SomeClass();
List<string> data = someObject.GetData();
data.Clear();
data.Add("you've been hacked");
```

This class is violating collection encapsulation principle

Outer caller can change its internal data

Assumptions About Previous Experience

There will be a module on performance testing

We will measure time and space requirements of a class

Get acquainted with asymptotic complexity

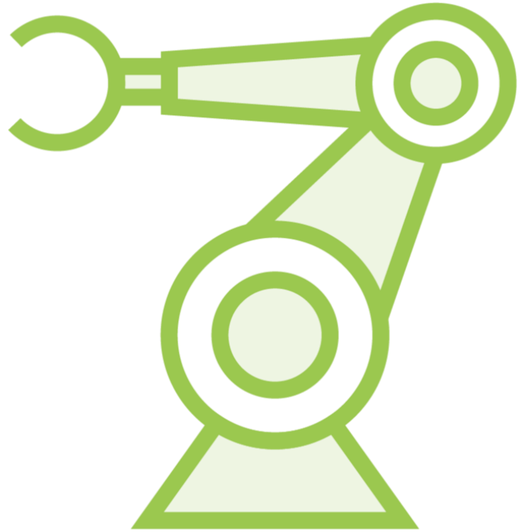
$O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$
a.k.a. constant, logarithmic, linear and log-linear time/space

This course is teaching production-grade designs

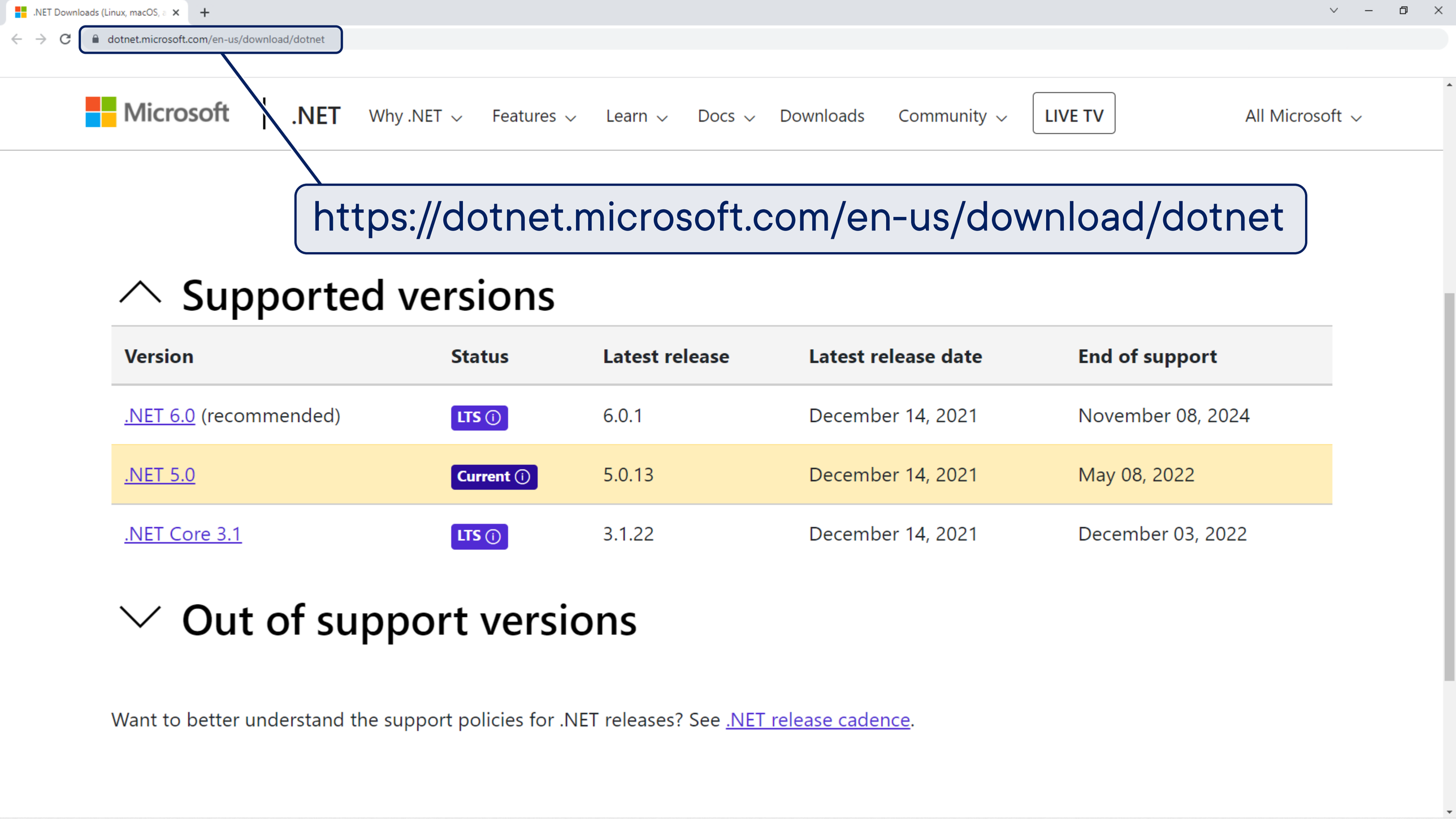
Classes from all demos can be applied to real problems

Preparing the Demos

Walk through the demo domain model



Install tools: .NET SDK and Visual Studio Code



Supported versions

Version	Status	Latest release	Latest release date	End of support
.NET 6.0 (recommended)	LTS	6.0.1	December 14, 2021	November 08, 2024
.NET 5.0	Current	5.0.13	December 14, 2021	May 08, 2022
.NET Core 3.1	LTS	3.1.22	December 14, 2021	December 03, 2022

Out of support versions

Want to better understand the support policies for .NET releases? See .NET release cadence.

Download Visual Studio Code -

code.visualstudio.com/download

Visual Studio Code | Docs | Updates | Blog | API | Extensions | FAQ | Learn | Search Docs | Download

https://code.visualstudio.com/download

Windows

Windows 7, 8, 10, 11

User Installer 64 bit 32 bit ARM

System Installer 64 bit 32 bit ARM

.zip 64 bit 32 bit ARM

.deb

Debian, Ubuntu

.rpm

Red Hat, Fedora, SUSE

.deb 64 bit ARM ARM 64

.rpm 64 bit ARM ARM 64

.tar.gz 64 bit ARM ARM 64

Snap Store

Mac

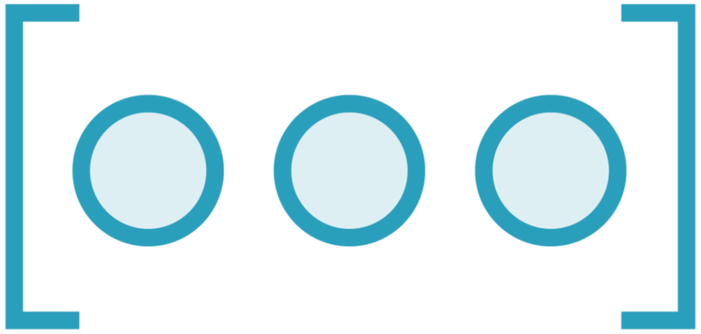
macOS 10.11+

.zip Universal Intel Chip Apple Silicon

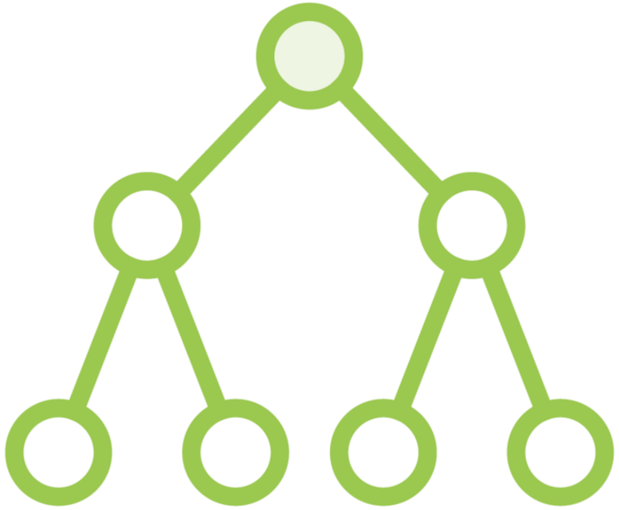
```
SomeClass obj = new SomeClass();

Func<SomeClass> f =
  () => new SomeClass();

List<SomeClass> list =
  new List<SomeClass>();

obj.Clear();

list.Clear();
```

- Everything is an object
- Even a function is an object!
- Multiple objects are an object, too
- An operation can apply to a single object
- But it can also apply to a collection
- Object-oriented programming erases distinction between one and many objects
- External interface becomes simple
- Intrinsic complexity becomes collection's problem to cope with

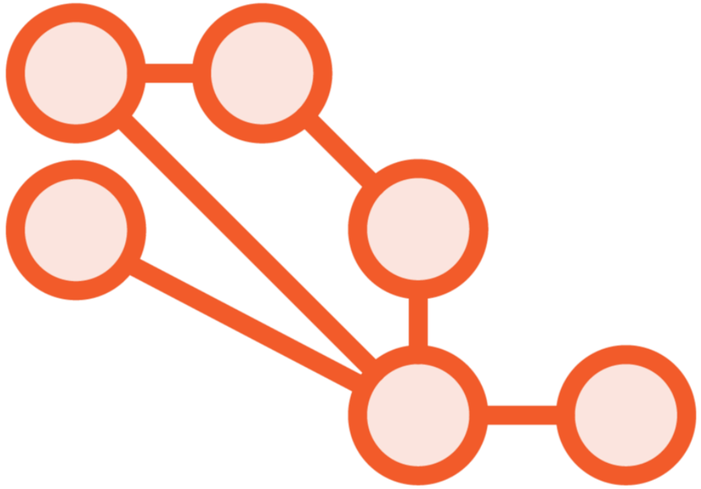
Logical Division of Collections



Linear
Array, list,
stack, queue, etc.

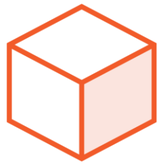


Associative
Dictionary,
hash set, etc.



Graphs
Proper graphs,
trees, etc.

Guidelines to Keep in Mind



Simplicity is your friend

We will favor linear collection over any other

If there is a solution based on a list or a stack – take it

We will favor associative collections over graphs

Above all, we will favor an encapsulated collection

Assumptions About Previous Experience

We will use many collections

Common array, `List`, `LinkedList`, `Stack`, `Queue`, `Dictionary`, `SortedDictionary`, `HashSet`, **etc.**

`IList<T>`
`List`, `ImmutableList`, `ImmutableArray`, **etc.**

`IList<T>` **is** *the* **linear collection in .NET**

`IDictionary<TKey, TValue>`
`Dictionary`, `SortedList`, `SortedDictionary`, **etc.**

`IDictionary<TKey, TValue>` **is** *the* **associative collection in .NET**

Comparing Linear to Associative Collections

List implementations

Stores items one after another

Can achieve constant-time retrieval

Random insert/retrieve may take linear time

Insert/remove on the last item usually guaranteed in constant time

Holds all items, even duplicates

Dictionary implementations

Associates each item with a key

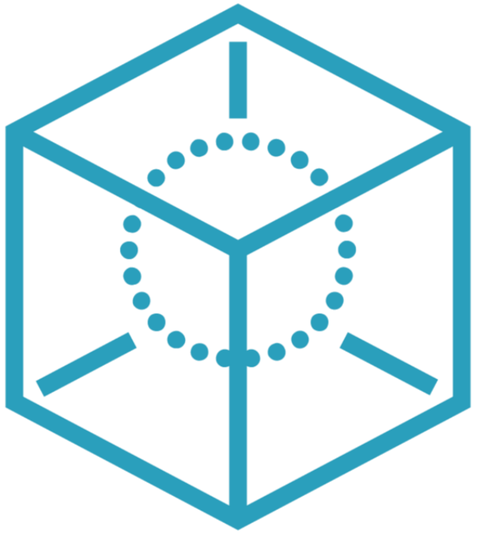
Can achieve constant-time retrieval

Random insert/retrieve takes the same time for all items

Adding/removing the last item takes the same time as for any other

Can only hold items with unique keys

Best Practices Working With Collections

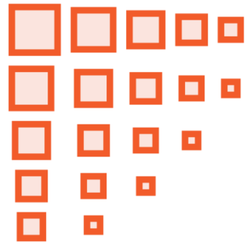


Favor encapsulated collection over any other

Hide collections behind an abstract interface

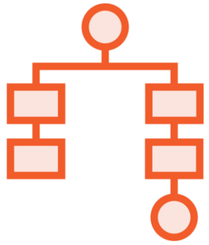
The best collection type is
the one you do not depend on

Understanding Generics



Collections are organically generic

But generics are applicable to other designs, too



Only statically typed languages require generics

```
Juice MakeJuice(Apples apples)
{
    Juice result = new();
    foreach (Apple apple in apples)
    {
        result.Add(apple.MakeJuice());
    }
    return result;
}
```

- Some algorithms only apply to one data type

```
IEnumerable<type> Sort<type>(type[] data)
{
    ...
    data[i].CompareTo(data[j]) ...
    ...
}
```

- Others can apply to more than one type
- Objects of the target type must be comparable

```
Apple[] crate = ...;
IEnumerable<Apple> apples =
    Sort<Apple>(crate);
```

- Algorithm will only specialize to a concrete type later, when used
- Apples must be mutually comparable

```
IComparable obj = new SomeType();

int c = obj.CompareTo(new OtherType());

void Sort(data)
{
    data[i].Compare(data[j])...
    data[i] = data[j]...
}

void Sort<T>(T[] data)
    where T : IComparable<T>
{
    data[i].CompareTo(data[j])...
}
```

- Compiler verifies that assignment is legal
- Static typing ensures that calls will be legal
- How to verify operations and assignments in a general algorithm?

- Use yet-unknown type T (type parameter)
- Specify any constraints
- Compiler can verify assignments and calls

```
public class List<T>
{
    ...
}
```

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

- Most collections are natively generic
- Same principles and data structures apply to any underlying element type
- Element type is required to verify assignments and calls
- Concrete element type is only specified when generic type is used
- Only constraints are verified on instantiation

Summary

Putting collections in perspective of domain operations

- Maintaining universal view on all things being objects

Outlined two major sorts of collections

- List and an associative list
- Graph structures do not generalize well

Summary

Related generics with collections

- Subsequent demos will build on this fundamental interconnectedness

Demonstrated reason to use a collection

Up Next: Applying linear collections