Loops and Collections

1

OUTLINE

- Arrays
- Foreach loops
- For loops
- While loops
- Do loops
- Generic collections
- List<T>
- Dictionary<TKey, TValue>
- Collection operators

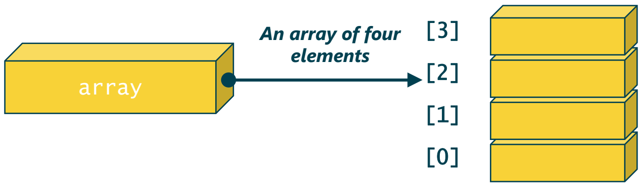
2

2

ARRAYS

An array is a collection of variables all of the same type

- Each element in the array can hold a single item
- Array elements are accessed by a zero-based index number in square brackets



```
int[] arr1 = new int[4]; // 0, 0, 0, 0
Console.WriteLine(arr1[0]); // 0
int[] arr2 = new int[] { 1, 3, 5, 7, 9 }; // array initializer
int x = arr2[2];
Console.WriteLine(x);// 5
```

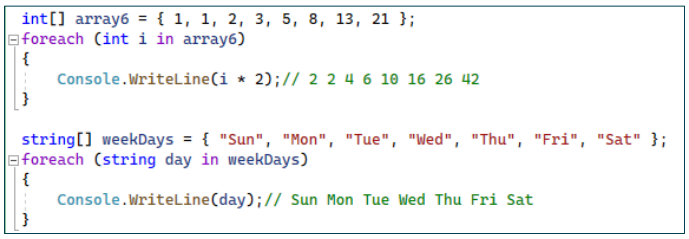
ARRAY INITIALISATION

- An array can be initialised without **new[ ]** if the type is defined and the values are provided
- An array can be *implicitly typed* using **new[ ]** and providing values whose type can be inferred
- When *declaring* and *initialising* an array variable separately, you must use the **new** operator

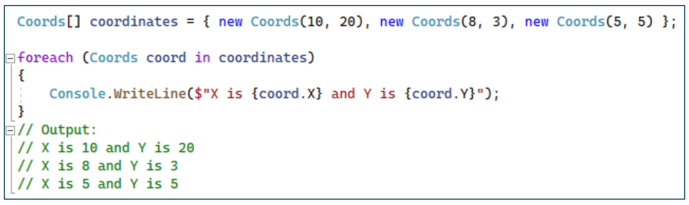
```
int[] arr3 = { 2, 4, 6, 8 }; // array initializer without new[]
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
Car[] cars = { new Car(), new Car(), new Car() };

// Implicitly typed arrays
var a = new[] { 1, 10, 100, 1000 }; // int[]
var b = new[] { "hello", null, "world" }; // string[]

// must use 'new' when declaring and initializing separately
int[] arr4;
arr4 = new int[] { 1, 3, 5, 7, 9 };   // OK
//arr4 = {1, 3, 5, 7, 9};   // Error
int[] arr5;
arr5 = new[] { 1, 3, 5, 7, 9 };   // OK
```

LOOPING THROUGH AN ARRAY

- The **foreach** statement enumerates the elements of a collection and executes its body for each element



```
int[] array6 = { 1, 1, 2, 3, 5, 8, 13, 21 };
foreach (int i in array6)
{
    Console.WriteLine(i * 2);// 2 2 4 6 10 16 26 42
}

string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
foreach (string day in weekDays)
{
    Console.WriteLine(day);// Sun Mon Tue Wed Thu Fri Sat
}
```



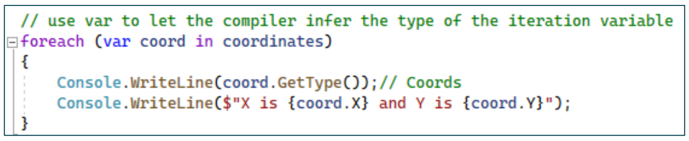
```
Coords[] coordinates = { new Coords(10, 20), new Coords(8, 3), new Coords(5, 5) };

foreach (Coords coord in coordinates)
{
    Console.WriteLine($"X is {coord.X} and Y is {coord.Y}");
}
// Output:
// X is 10 and Y is 20
// X is 8 and Y is 3
// X is 5 and Y is 5
```

5

5

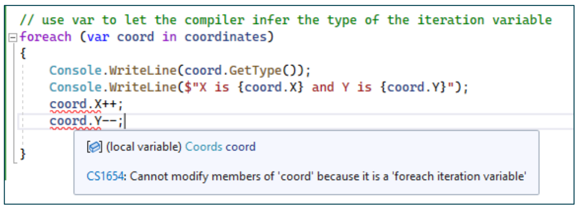
FOREACH

- You can use **var** in the **foreach** loop to let the compiler infer the type of the iteration variable



```
// use var to let the compiler infer the type of the iteration variable
foreach (var coord in coordinates)
{
    Console.WriteLine(coord.GetType());// Coords
    Console.WriteLine($"X is {coord.X} and Y is {coord.Y}");
}
```

- You cannot modify the members of the iteration variable within a **foreach** loop



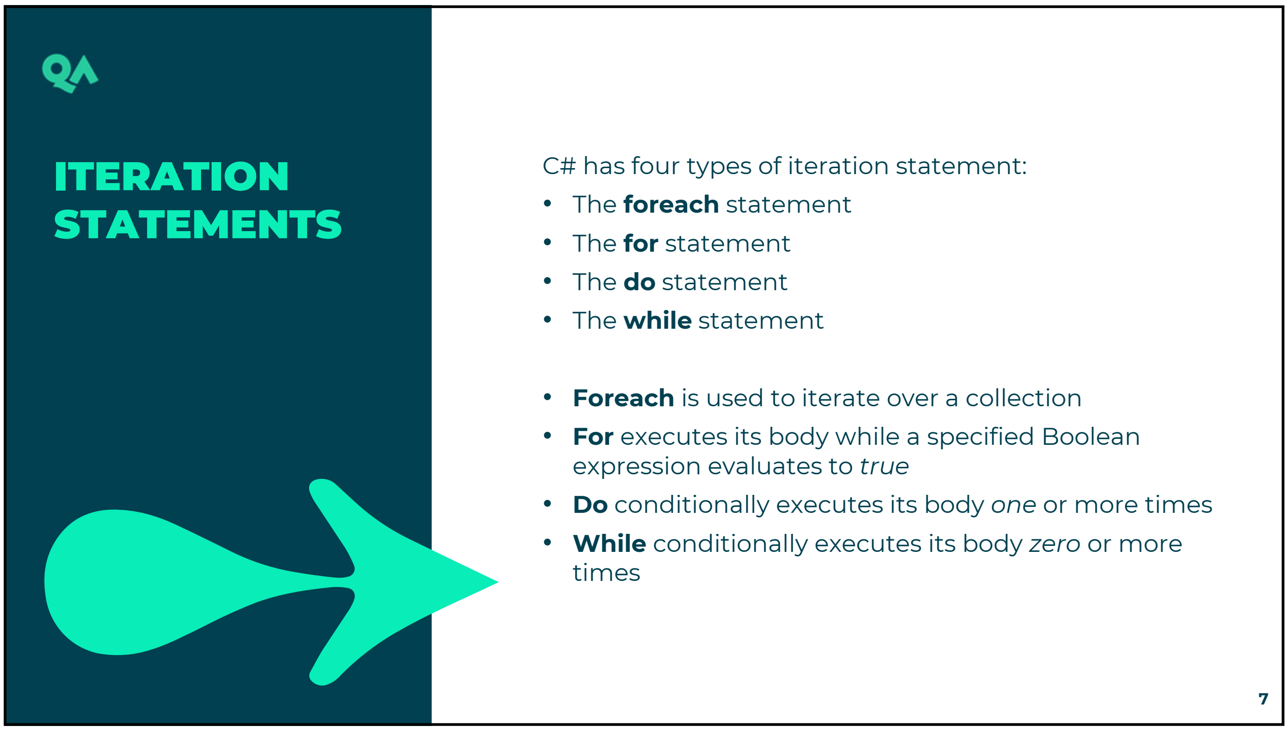
```
// use var to let the compiler infer the type of the iteration variable
foreach (var coord in coordinates)
{
    Console.WriteLine(coord.GetType());
    Console.WriteLine($"X is {coord.X} and Y is {coord.Y}");
    coord.X++;
    coord.Y--;
}
```

(local variable) Coords coord

CS1654: Cannot modify members of 'coord' because it is a 'foreach iteration variable'

6

6



ITERATION STATEMENTS

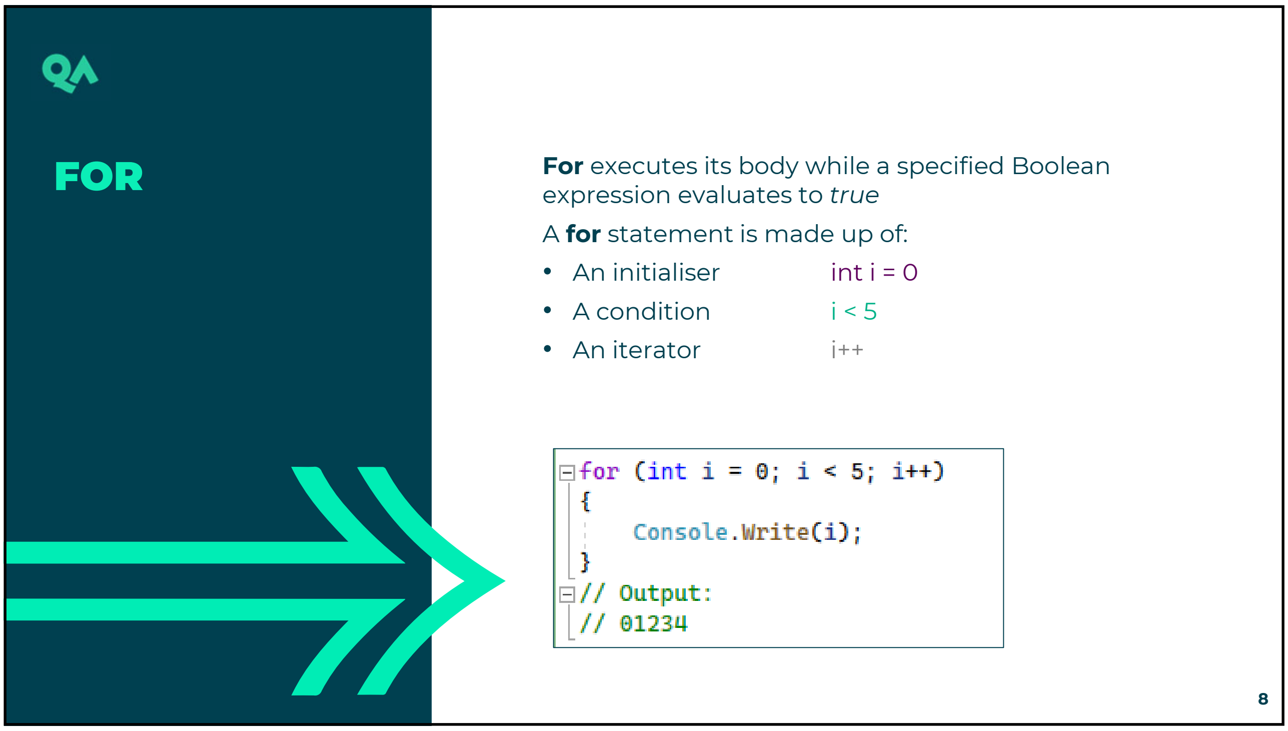
C# has four types of iteration statement:

- The **foreach** statement
- The **for** statement
- The **do** statement
- The **while** statement

- **Foreach** is used to iterate over a collection
- **For** executes its body while a specified Boolean expression evaluates to *true*
- **Do** conditionally executes its body *one* or more times
- **While** conditionally executes its body *zero* or more times

7

7



FOR

For executes its body while a specified Boolean expression evaluates to *true*

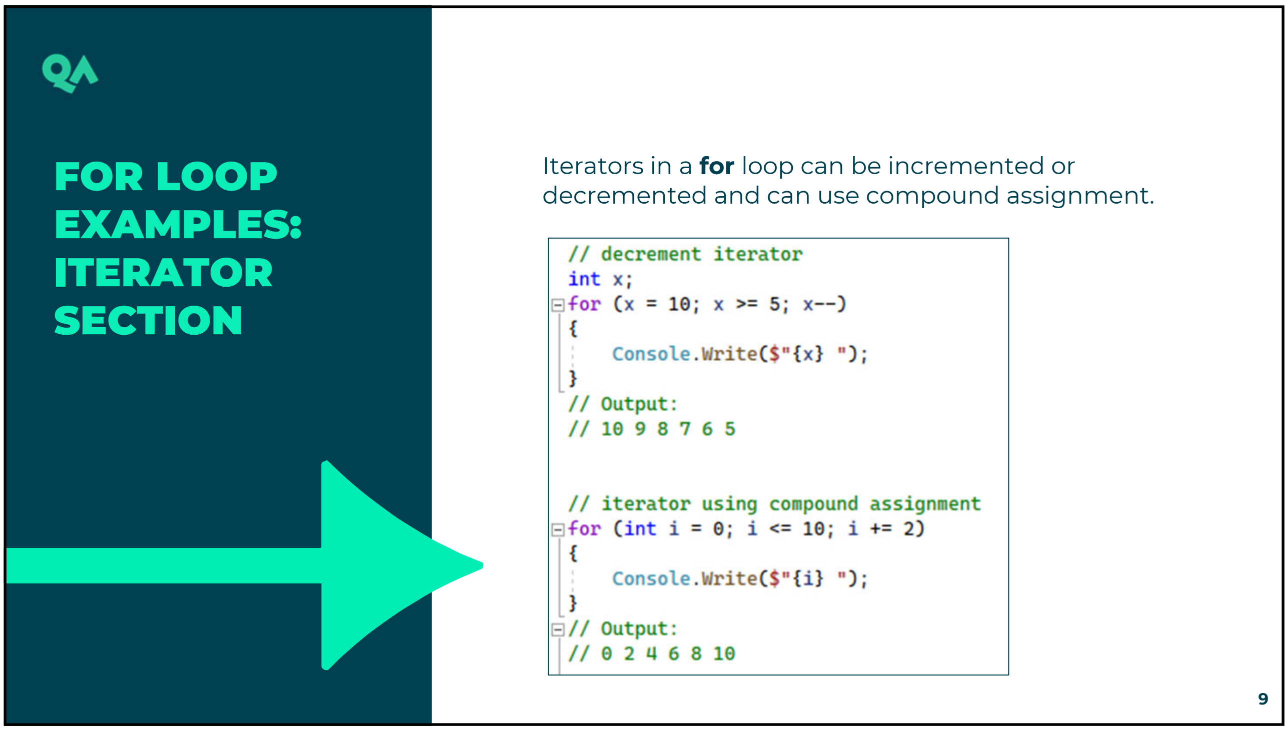
A **for** statement is made up of:

• An initialiser	int i = 0
• A condition	i < 5
• An iterator	i++

```
for (int i = 0; i < 5; i++)
{
    Console.Write(i);
}
// Output:
// 01234
```

8

8



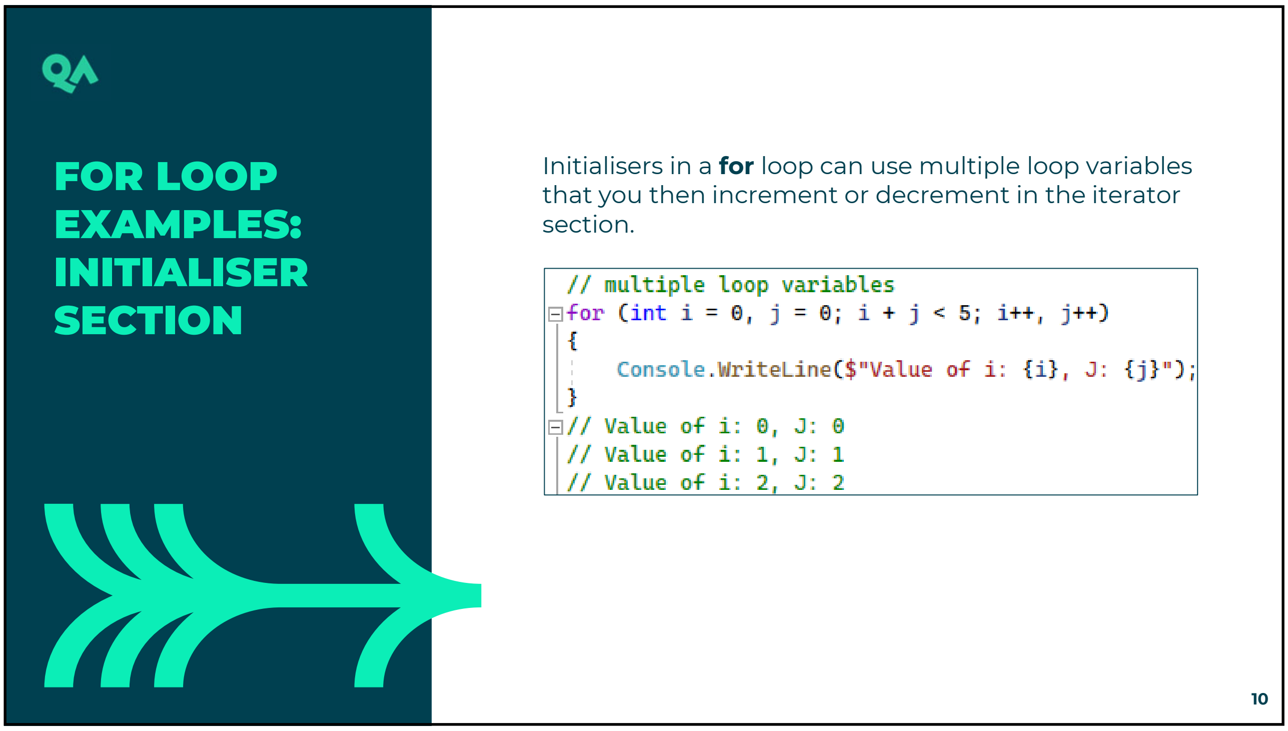
FOR LOOP EXAMPLES: ITERATOR SECTION

Iterators in a **for** loop can be incremented or decremented and can use compound assignment.

```
// decrement iterator
int x;
for (x = 10; x >= 5; x--)
{
    Console.Write($"{x} ");
}
// Output:
// 10 9 8 7 6 5


// iterator using compound assignment
for (int i = 0; i <= 10; i += 2)
{
    Console.Write($"{i} ");
}
// Output:
// 0 2 4 6 8 10
```

9

9



FOR LOOP EXAMPLES: INITIALISER SECTION

Initialisers in a **for** loop can use multiple loop variables that you then increment or decrement in the iterator section.

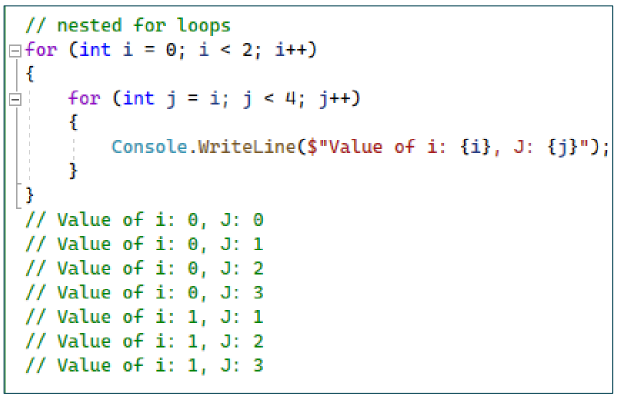
```
// multiple loop variables
for (int i = 0, j = 0; i + j < 5; i++, j++)
{
    Console.WriteLine($"Value of i: {i}, J: {j}");
}
// Value of i: 0, J: 0
// Value of i: 1, J: 1
// Value of i: 2, J: 2
```

10

10

FOR LOOP EXAMPLES: NESTED LOOPS

You can nest **for** loops:



```
// nested for loops
for (int i = 0; i < 2; i++)
{
    for (int j = i; j < 4; j++)
    {
        Console.WriteLine($"Value of i: {i}, J: {j}");
    }
}
// Value of i: 0, J: 0
// Value of i: 0, J: 1
// Value of i: 0, J: 2
// Value of i: 0, J: 3
// Value of i: 1, J: 1
// Value of i: 1, J: 2
// Value of i: 1, J: 3
```

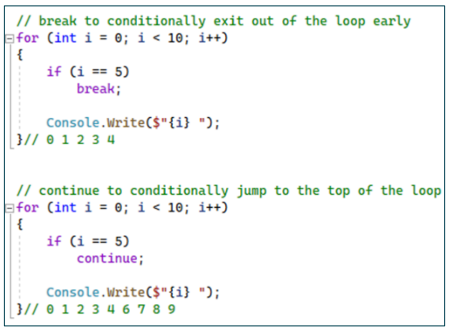
11

11

FOR LOOP EXAMPLES: JUMP STATEMENTS

C# has the following jump statements that are used with loops:

- **continue:** Start a new iteration
- **break:** Terminate the closest enclosing loop statement
- **goto:** Transfer control to a statement that is marked by a label



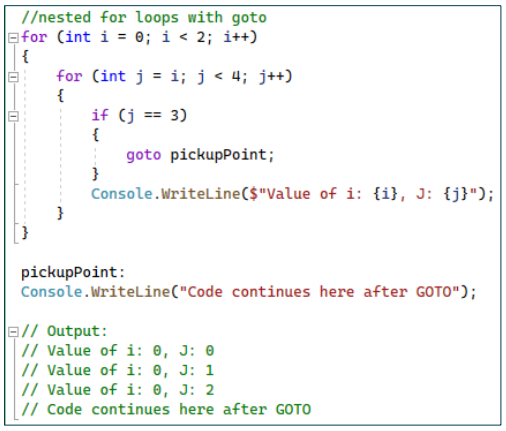
```
// break to conditionally exit out of the loop early
for (int i = 0; i < 10; i++)
{
    if (i == 5)
        break;

    Console.Write($"{i} ");
}// 0 1 2 3 4


// continue to conditionally jump to the top of the loop
for (int i = 0; i < 10; i++)
{
    if (i == 5)
        continue;

    Console.Write($"{i} ");
}// 0 1 2 3 4 6 7 8 9
```

12

12

FOR LOOP EXAMPLES: JUMP STATEMENTS

The **goto** statement transfers control to a statement that is marked by a label which can be used to exit out of nested loops:



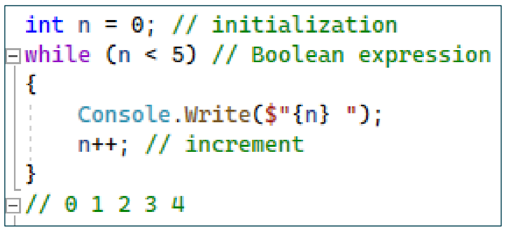
```
//nested for loops with goto
for (int i = 0; i < 2; i++)
{
    for (int j = i; j < 4; j++)
    {
        if (j == 3)
        {
            goto pickupPoint;
        }
        Console.WriteLine($"Value of i: {i}, J: {j}");
    }
}

pickupPoint:
Console.WriteLine("Code continues here after GOTO");

// Output:
// Value of i: 0, J: 0
// Value of i: 0, J: 1
// Value of i: 0, J: 2
// Code continues here after GOTO
```

13

13

WHILE LOOPS

A **while** loop conditionally executes its body *zero* or more times whilst a specified Boolean expression evaluates to *true.*



```
int n = 0; // initialization
while (n < 5) // Boolean expression
{
    Console.Write($"{n} ");
    n++; // increment
}
// 0 1 2 3 4
```

It is important to modify the conditional variable otherwise you will have an infinite loop.

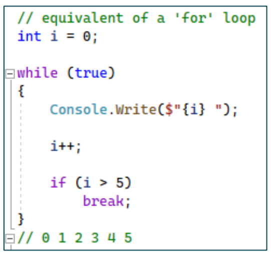
14

14

WHILE LOOP EXAMPLES

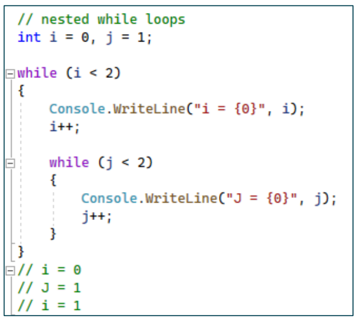
A **while** loop is typically used when you don't know how many times you want the loop body to execute, which is why it uses a conditional test.

A **for** loop is used when you can count the number of times you want the loop body to execute.



```
// equivalent of a 'for' loop
int i = 0;

while (true)
{
    Console.Write($"{i} ");

    i++;

    if (i > 5)
        break;
}
// 0 1 2 3 4 5
```

While loops can be nested:



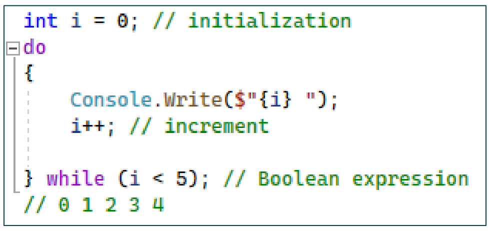
```
// nested while loops
int i = 0, j = 1;

while (i < 2)
{
    Console.WriteLine("i = {0}", i);
    i++;

    while (j < 2)
    {
        Console.WriteLine("J = {0}", j);
        j++;
    }
}
// i = 0
// J = 1
// i = 1
```

15

15

DO LOOPS

A **do** loop conditionally executes its body *one* or more times whilst a specified Boolean expression evaluates to *true.*



```
int i = 0; // initialization
do
{
    Console.Write($"{i} ");
    i++; // increment

} while (i < 5); // Boolean expression
// 0 1 2 3 4
```

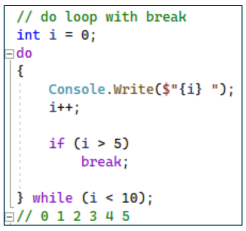
It is important to modify the conditional variable, otherwise you will have an infinite loop.

16

16

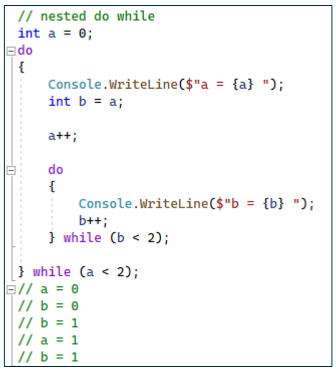
DO LOOP EXAMPLES

A **do** loop can use the jump statements **continue**, **break,** and **goto:**



```
// do loop with break
int i = 0;
do
{
    Console.Write($"{i} ");
    i++;

    if (i > 5)
        break;

} while (i < 10);
// 0 1 2 3 4 5
```

Do loops can be nested:

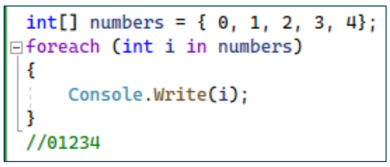


```
// nested do while
int a = 0;
do
{
    Console.WriteLine($"a = {a} ");
    int b = a;

    a++;

    do
    {
        Console.WriteLine($"b = {b} ");
        b++;
    } while (b < 2);

} while (a < 2);
// a = 0
// b = 0
// b = 1
// a = 1
// b = 1
```

17

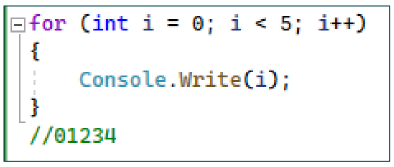
17

LOOPS SUMMARY



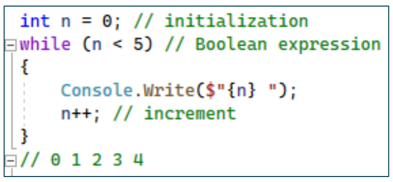
```
int[] numbers = { 0, 1, 2, 3, 4};
foreach (int i in numbers)
{
    Console.Write(i);
}
//01234
```

foreach



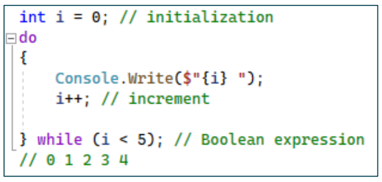
```
for (int i = 0; i < 5; i++)
{
    Console.Write(i);
}
//01234
```

for



```
int n = 0; // initialization
while (n < 5) // Boolean expression
{
    Console.Write($"{n} ");
    n++; // increment
}
// 0 1 2 3 4
```

while



```
int i = 0; // initialization
do
{
    Console.Write($"{i} ");
    i++; // increment

} while (i < 5); // Boolean expression
// 0 1 2 3 4
```

do

18

18

GENERIC COLLECTIONS

The **System.Collections.Generic** namespace contains classes and interfaces that define generic collections.

A **generic collection** allows users to create strongly typed collections that provide better performance and type safety than non-generic collections.

Common generic collection classes are:

- **List<T>**
- **Dictionary<TKey, TValue>**

- <T> is the type of elements in the list
- <TKey> is the type of the keys in the dictionary
- <TValue> is the type of the values in the dictionary

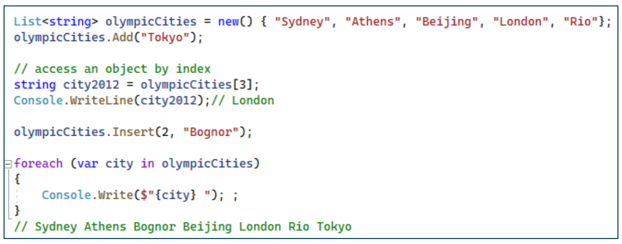
19

19

LIST<T>

The **List<T>** class represents a strongly typed list of objects

- The objects can be accessed by index
- The list can be manipulated (insert, add, remove)
- The list can be searched
- The list can be sorted

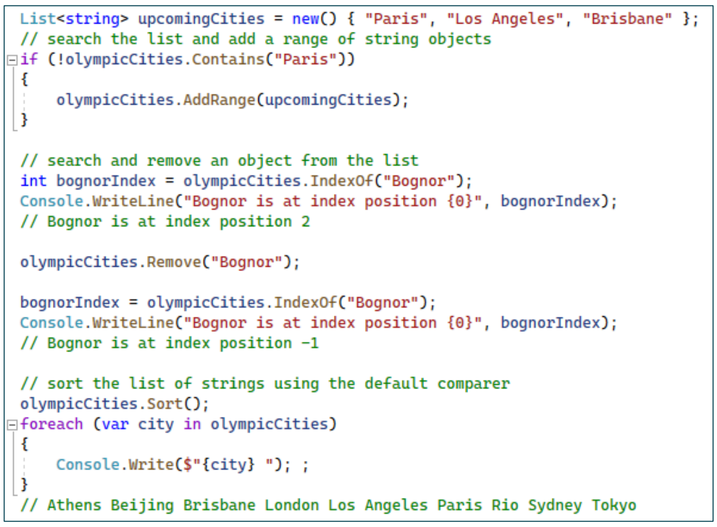


```
List<string> olympicCities = new() { "Sydney", "Athens", "Beijing", "London", "Rio"};
olympicCities.Add("Tokyo");

// access an object by index
string city2012 = olympicCities[3];
Console.WriteLine(city2012);// London

olympicCities.Insert(2, "Bognor");

foreach (var city in olympicCities)
{
    Console.Write($"{city} "); ;
}
// Sydney Athens Bognor Beijing London Rio Tokyo
```

20

20

LIST<T> EXAMPLE



```
List<string> upcomingCities = new() { "Paris", "Los Angeles", "Brisbane" };
// search the list and add a range of string objects
if (!olympicCities.Contains("Paris"))
{
    olympicCities.AddRange(upcomingCities);
}

// search and remove an object from the list
int bognorIndex = olympicCities.IndexOf("Bognor");
Console.WriteLine("Bognor is at index position {0}", bognorIndex);
// Bognor is at index position 2

olympicCities.Remove("Bognor");

bognorIndex = olympicCities.IndexOf("Bognor");
Console.WriteLine("Bognor is at index position {0}", bognorIndex);
// Bognor is at index position -1

// sort the list of strings using the default comparer
olympicCities.Sort();
foreach (var city in olympicCities)
{
    Console.Write($"{city} "); ;
}
// Athens Beijing Brisbane London Los Angeles Paris Rio Sydney Tokyo
```

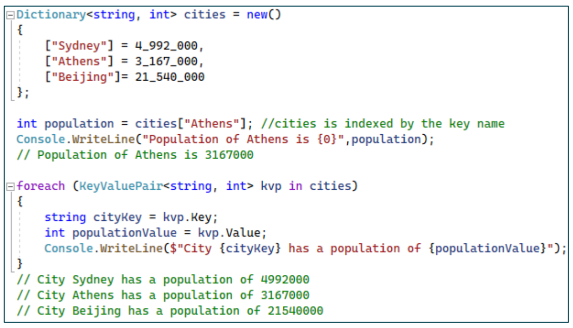
21

21

DICTIONARY <TKEY,TVALUE>

The **Dictionary<TKey,TValue>** class represents a strongly typed collection of *keys* and *values*

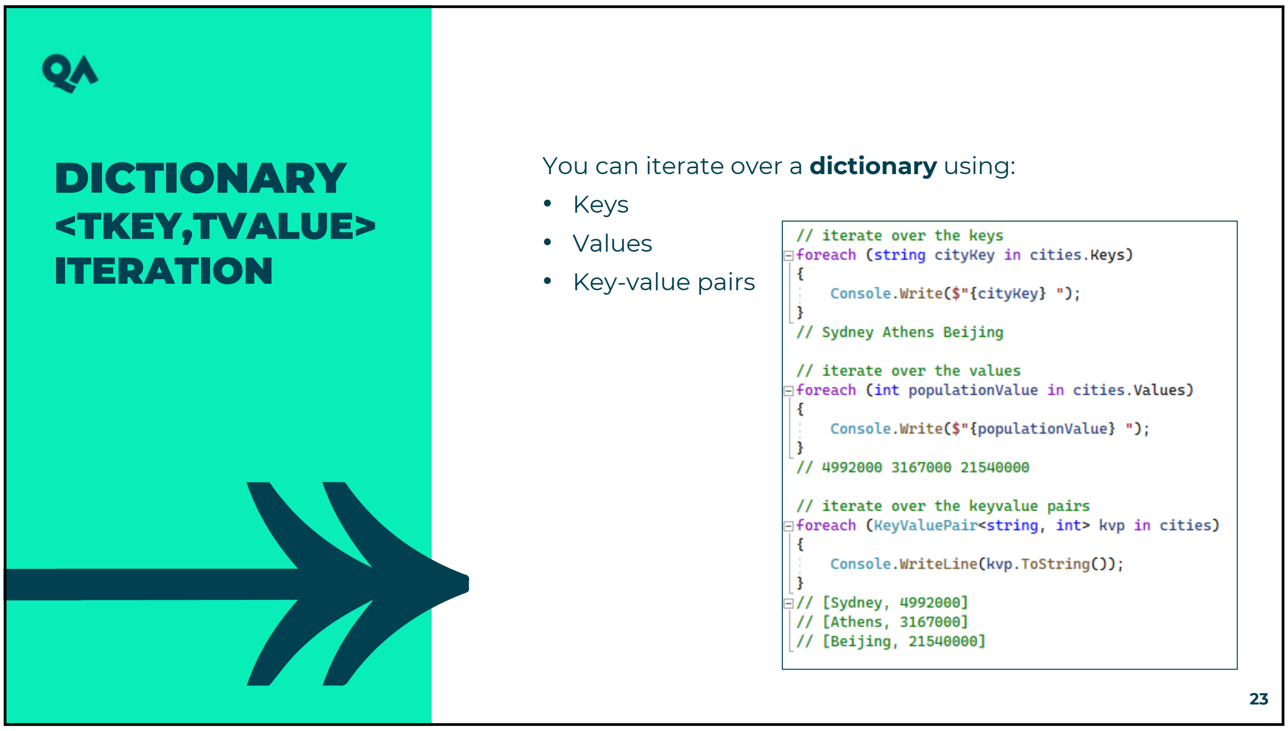
- The keys must be unique and cannot be null
- The values can be null or duplicates
- The values are accessed by indexing the key in square brackets **[key]**



```
Dictionary<string, int> cities = new()
{
    ["Sydney"] = 4_992_000,
    ["Athens"] = 3_167_000,
    ["Beijing"]= 21_540_000
};

int population = cities["Athens"]; //cities is indexed by the key name
Console.WriteLine("Population of Athens is {0}",population);
// Population of Athens is 3167000

foreach (KeyValuePair<string, int> kvp in cities)
{
    string cityKey = kvp.Key;
    int populationValue = kvp.Value;
    Console.WriteLine($"City {cityKey} has a population of {populationValue}");
}
// City Sydney has a population of 4992000
// City Athens has a population of 3167000
// City Beijing has a population of 21540000
```

22

22



DICTIONARY <TKEY,TVALUE> ITERATION

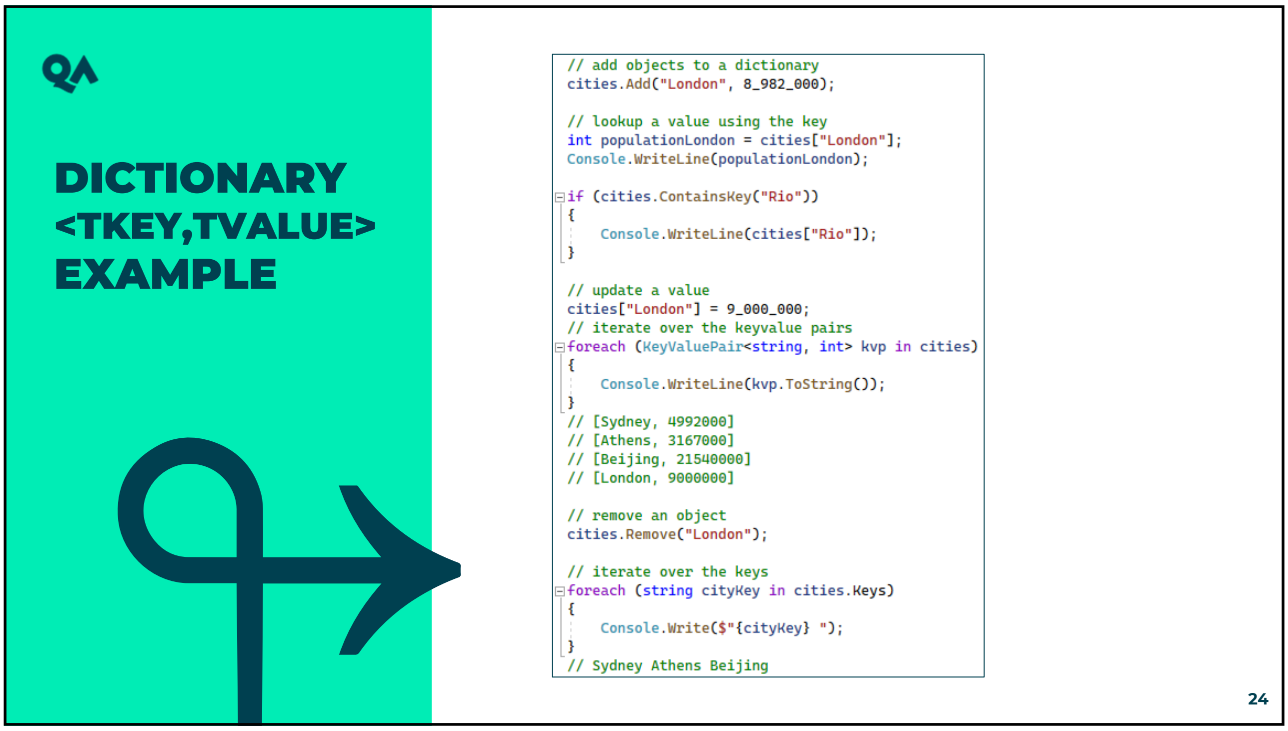
You can iterate over a **dictionary** using:

- Keys
- Values
- Key-value pairs

```
// iterate over the keys
foreach (string cityKey in cities.Keys)
{
    Console.Write($"{cityKey} ");
}
// Sydney Athens Beijing

// iterate over the values
foreach (int populationValue in cities.Values)
{
    Console.Write($"{populationValue} ");
}
// 4992000 3167000 21540000

// iterate over the keyvalue pairs
foreach (KeyValuePair<string, int> kvp in cities)
{
    Console.WriteLine(kvp.ToString());
}
// [Sydney, 4992000]
// [Athens, 3167000]
// [Beijing, 21540000]
```

23

23

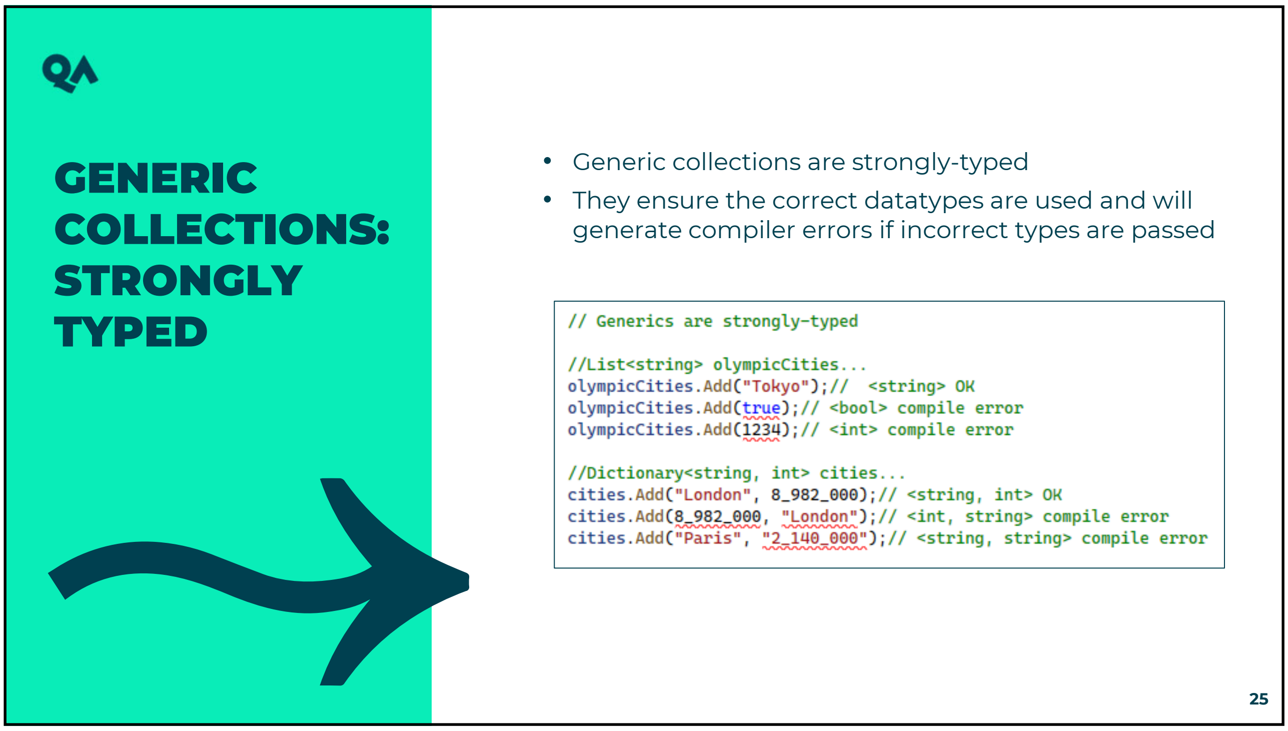


DICTIONARY <TKEY,TVALUE> EXAMPLE

```
// add objects to a dictionary
cities.Add("London", 8_982_000);

// lookup a value using the key
int populationLondon = cities["London"];
Console.WriteLine(populationLondon);

if (cities.ContainsKey("Rio"))
{
    Console.WriteLine(cities["Rio"]);
}

// update a value
cities["London"] = 9_000_000;
// iterate over the keyvalue pairs
foreach (KeyValuePair<string, int> kvp in cities)
{
    Console.WriteLine(kvp.ToString());
}
// [Sydney, 4992000]
// [Athens, 3167000]
// [Beijing, 21540000]
// [London, 9000000]

// remove an object
cities.Remove("London");

// iterate over the keys
foreach (string cityKey in cities.Keys)
{
    Console.Write($"{cityKey} ");
}
// Sydney Athens Beijing
```

24

24



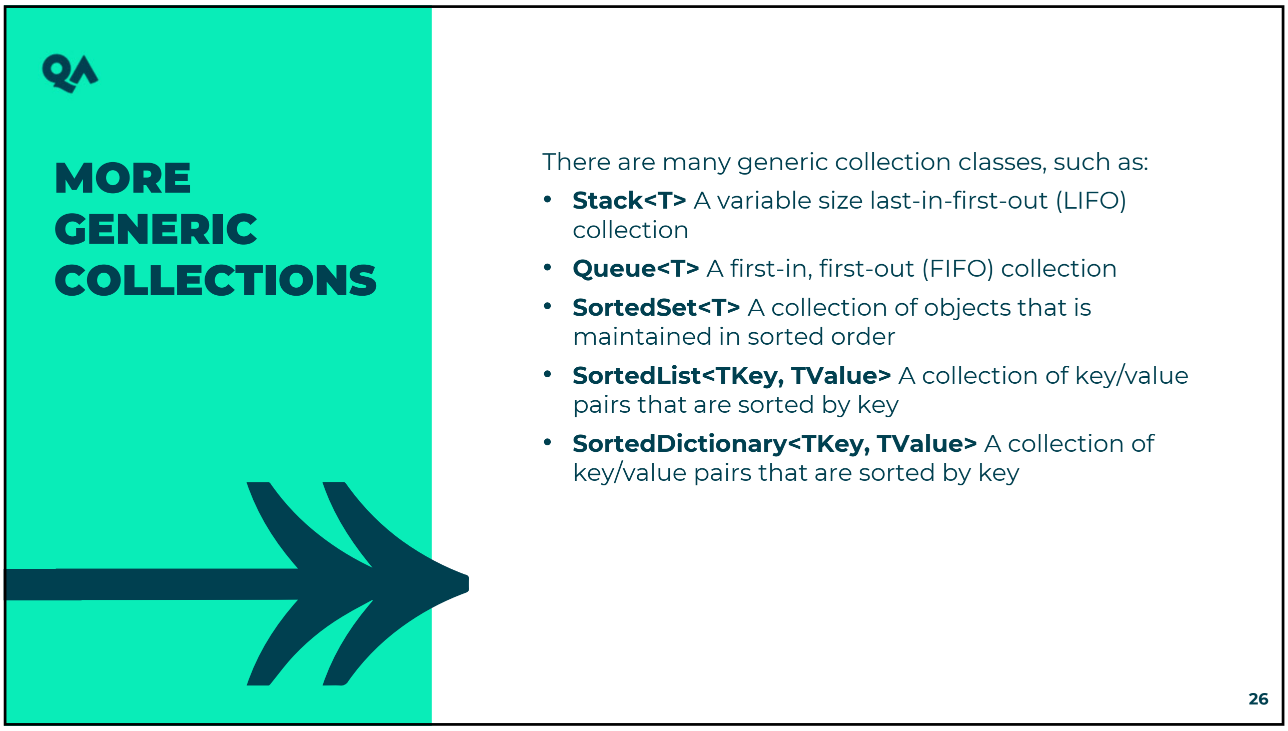
GENERIC COLLECTIONS: STRONGLY TYPED

- Generic collections are strongly-typed
- They ensure the correct datatypes are used and will generate compiler errors if incorrect types are passed

```
// Generics are strongly-typed

//List<string> olympicCities...
olympicCities.Add("Tokyo");//  <string> OK
olympicCities.Add(true);// <bool> compile error
olympicCities.Add(1234);// <int> compile error

//Dictionary<string, int> cities...
cities.Add("London", 8_982_000);// <string, int> OK
cities.Add(8_982_000, "London");// <int, string> compile error
cities.Add("Paris", "2_140_000");// <string, string> compile error
```

25

25



MORE GENERIC COLLECTIONS

There are many generic collection classes, such as:

- **Stack<T>** A variable size last-in-first-out (LIFO) collection
- **Queue<T>** A first-in, first-out (FIFO) collection
- **SortedSet<T>** A collection of objects that is maintained in sorted order
- **SortedList<TKey, TValue>** A collection of key/value pairs that are sorted by key
- **SortedDictionary<TKey, TValue>** A collection of key/value pairs that are sorted by key

26

26

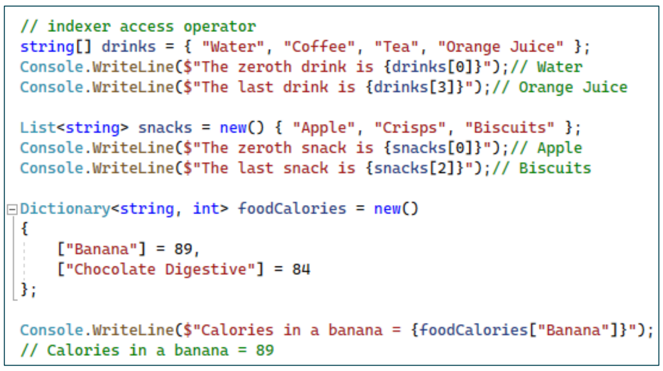
COLLECTION OPERATORS

There are three useful operators for working with collections:

- Array element or indexer access operator **[ ]**
- Index from end operator **^**
- Range operator **..**

INDEXER ACCESS OPERATOR

- The array element or indexer access operator **[ ]** is used to access elements in an array or collection using an index value or key



```
// indexer access operator
string[] drinks = { "Water", "Coffee", "Tea", "Orange Juice" };
Console.WriteLine($"The zeroth drink is {drinks[0]}");// Water
Console.WriteLine($"The last drink is {drinks[3]}");// Orange Juice

List<string> snacks = new() { "Apple", "Crisps", "Biscuits" };
Console.WriteLine($"The zeroth snack is {snacks[0]}");// Apple
Console.WriteLine($"The last snack is {snacks[2]}");// Biscuits

Dictionary<string, int> foodCalories = new()
{
    ["Banana"] = 89,
    ["Chocolate Digestive"] = 84
};

Console.WriteLine($"Calories in a banana = {foodCalories["Banana"]}");
// Calories in a banana = 89
```

INDEX FROM END OPERATOR AND RANGE OPERATOR

- The **index from end** operator **^** indicates the element position from the end of a sequence
- The **range** operator **..** specifies the start and end of a range of indices
  - The left-hand operand is inclusive
  - The right-hand operand is exclusive

```
List<int> numbers = new() { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
// index from end operator
var firstFromEnd_10 = numbers[^1];
var thirdFromEnd_8 = numbers[^3];

// range operator
var slice_345678910 = numbers.ToArray()[2..];
var slice_12345678910 = numbers.ToArray()[..];

// range and index from end operators
var slice_34567 = numbers.ToArray()[2..^3];
var slice_1234567 = numbers.ToArray()[..^3];
```

29

29

SUMMARY

- Arrays
- Foreach loops
- For loops
- While loops
- Do loops
- Generic collections
- List<T>
- Dictionary<TKey, TValue>
- Collection operators

30

30

ACTIVITY:
Exercise 5

31

31