C# FUNDAMENTALS

SESSIE 3/4

PLAN DAG 3

- Arrays
- Generics
- Optional: Indexers
- Collections
- Optional: multithreading & concurrent collections
- Foreach
- Assemblies
- Constants
- LINQ

ARRAYS

Multiple values of the same type in one variable

string[] cars = {"Volvo", "BMW", "Ford", "Mazda"};

Elements can be obtained by index, index starts counting at 0

cars[0] // volvo

cars[0] = “Tesla” //override the value of the first position

cars.Length //4

Array.Sort(cars) // alphabetically ordered

- Length needs to be clear upon declaration:
  - Int[] array = new int[60]
  - Int[] array2 = {1,2,3}

EXERCISE

- Create an array of strings of your colleagues
- Change the name of a colleague
- Sort them
- Print them
- Can you add one?

GENERICS

- Write and use parameterized types

- Avoids problems with unexpected input types during runtime

- You'll find them in the standard libraries and APIs a lot, but less common necessary to use for mainstream applications

GENERIC CLASSES

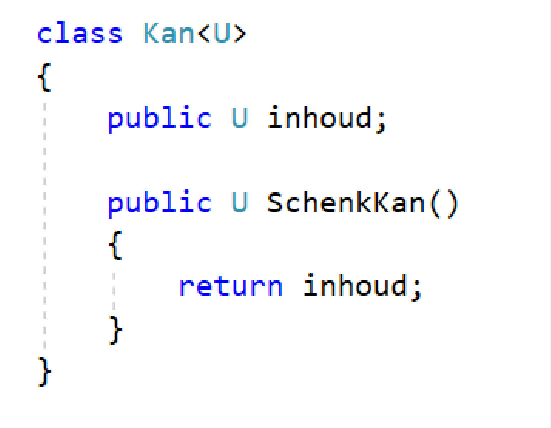
Add formal type parameter to class

```
public class Example<T>
{
        public T ExampleField{ get; set;}
}
```

Specify type when calling the class

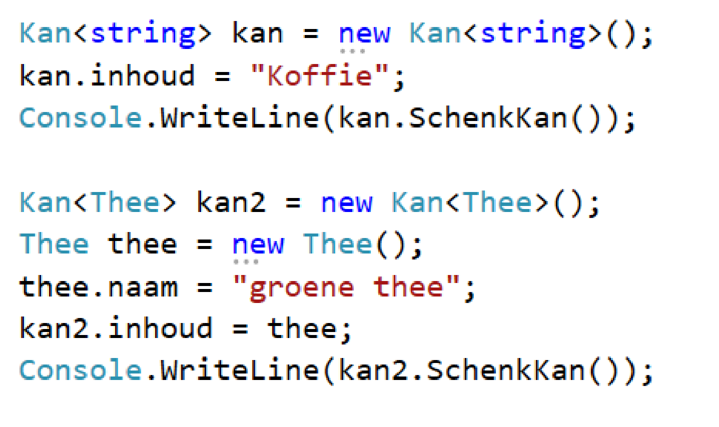
```
Example<Animal> ex = new Example<Animal>();
```

EXERCISE: CREATE A CLASS WITH GENERIC DATA TYPE



```
class Kan<U>
{
    public U inhoud;

    public U SchenkKan()
    {
        return inhoud;
    }
}
```

- Make a new class Bag that takes a generic type T
- Add a class Groceries
- In a main method, instantiate a bag and specify Groceries as the type
- Make an array of groceries and create a bag using these groceries
- Make a new method to unpack groceries that takes a bag as parameter an object of type Groceries



```
Kan<string> kan = new Kan<string>();
kan.inhoud = "Koffie";
Console.WriteLine(kan.SchenkKan());

Kan<Thee> kan2 = new Kan<Thee>();
Thee thee = new Thee();
thee.naam = "groene thee";
kan2.inhoud = thee;
Console.WriteLine(kan2.SchenkKan());
```

INDEXERS

- Special type of property
- Array of something that the class has a has-a relationship with
- Instances of a class have an indexer
- Indexer is used to access an instance of the has-a class calling the indexer with the index

```
class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];
   int nextIndex = 0;

   // Define the indexer to allow client code to use [] notation.
   public T this[int i] => arr[i];

   public void Add(T value)
   {
      if (nextIndex >= arr.Length)
         throw new IndexOutOfRangeException($"The collection can hold only {arr.Length} elements.");
      arr[nextIndex++] = value;
   }
}
```

EXERCISE

- Create a Box with a generic T

- Give it an indexer

- Add 10 animals to the box

- Get the animal at index 4 out

COLLECTIONS

- Array alternative for creating a group of objects
- Arrays for fixed number of objects
- Collections for flexible grouping of objects
- Collections are classes, and need to be instantiated before they can be used

- Collections in three namespaces:
  - System.Collections.Generic → used for storing one type of object
  - System.Collections.Concurrent → thread safe, should be used for accessing collections with multiple threads
  - System.Collections → stores all elements as objects of type System.Object, least preferred (not for today)

LIST

- List<T> class
- Stored in order they are added
- Items can be retrieved with the index

```
var colors = new List<string>();
colors.Add(“green");
colors.Add(“orange");
colors.Add("pink");
colors.Add(“blue");


for (var index = 0; index < colors.Count; index++)
{
    Console.Write(colors[index] + " ");
}

colors.Remove(“orange”); //or .RemoveAt(1)
```

EXERCISE

- Create a Customer class
- And create an address class
- Give the customer a list of addresses
- Create a method to fill the list of addresses with some data
- Also make sure you can modify a list by adding, removing and inserting something in the middle

DICTIONARY

- Groups of key – value pairs
- Keys must be unique
- Items are retrieved by their key
- Dictionary<Tkey, Tvalue>

```
var dict = new Dictionary<int, string>();
dict.Add(1,"One");
dict.Add(2,"Two");
dict.Add(3,"Three");
```

PROPERTIES AND METHODS ON DICTIONARIES

PROPS

- Count
- IsReadOnly
- Item(key) → get/set element with certain key
- Keys → returns collection of keys
- Values → returns collection of values

METHODS

- Add
- Remove() → removes the first
- Remove(key) → removes element with certain key
- ContainsKey / ContainsValue
- Clear() → removes all elements
- TryGetValue

EXERCISE

- Add a class order (continue from last exercise)
- Add a class products
- Create a dictionary that keeps orderID and a collection of the products ordered
- Create a method to add some data to the dictionary

SORTEDLIST

- Both generic and non-generic version
  - Generic: System.Collections.Generic
  - Non-generic: System.Collections
- Array of key/value pairs (SortedList is implemented as two internal arrays: keys and values)
- Elements can be retrieved as KeyValuePair<K,V> objects
- Elements are sorted by key

```
SortedList<int, string> mySortedList = new
SortedList<int,string>();

mySortedList.Add(1, “one”);

Console.WriteLine(mySortedList[1]); //use key

//Following will throw runtime exception:
KeyNotFoundException
Console.WriteLine(mySortedList[2]);
```

SORTEDLIST VS DICTIONARY

- Sortedlist by in ascending order of key

- Different way of memory management, dictionaries are objects all over the heap and dictionaries require more memory for storage

- Dictionary faster with adding and removing, sortedlist faster retrieving

EXERCISE

- Let’s see if we can measure the differences!
- Create a SortedList and SortedDictionary (special dictionary, sorted by key) with int keys and string values
- Add items to it in a loop and search for items in a loop
- Measure how long it takes to add large volumes (such as 10.000 or 100.000) values
- Do the same for searching

OBJECT-INITIALIZER SYNTAX

```
SortedList<int,string> sortedList1 = new SortedList<int,string>()
                        {
                            {3, "Three"},
                            {4, "Four"},
                            {1, "One"},
                            {5, "Five"},
                            {2, "Two"}
                        };
```

ICOMPARER<T> INTERFACE

- Has a method that compares two objects: Compare(Object, Object)
- Used for sorting

ICOMPARABLE<T> INTERFACE

- Has a method that compares one object to itself: CompareTo(Object)
- Used for sorting

IENUMERABLE<T> INTERFACE

- Exposes methods GetEnumerator that need to overriden in order to make iteration possible
- Implemented by collections in the System.Collections.Generic namespace (List a.o.)
- Collections that implement Ienumerable<t> can be iterated over using foreach

-ER VS –BLE INTERFACES

- -able usually used for classes to implement. E.g. a class that implements IComparable has a default order for sorting

- -er usually created on the spot and sent to a method that needs one. E.g. an implementation of IComparer can be used to give another sorting order than the default to a SortedList

QUEUE

- FIFO collection (first in, first out)
- Queue<T>
- Enqueue method to add, dequeue method to remove
- Peek method to have a look at the next item without dequeuing it
- ToArray → method to copy the elements of the queue to an array while making a copy of the queue in order to still have a queue after the operation

STACK

- Stack<T> class
- LIFO collection, last in firs out
- Push method to add to the stack
- Pop method to get from the stack
- Peek to look at the next item, without popping it
- ToArray → method to copy the elements of the stack to an array while making a copy of the stack with reversed order in order to still have a stack after the operation

FOREACH

```
var myNumbers = new List<int> { 0, 1, 2, 3};
int count = 0;
foreach (int element in myNumbers)
{
    count++;
    Console.WriteLine($"Element #{count}:
            {element}");
}
```

```
foreach(KeyValuePair<string, string> entry in
myDictionary)
{
    // do something with entry.Value or
            //entry.Key
}
```

FOREACH

- Foreach uses the IEnumarable and IEnumerator to go over all the elements in a collection

Compiled foreach statement looks like:

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

EXERCISE

- Use foreach to:
  - Make a method in addresses to find the address by zipcode from a list of addresses
  - Create a method in address to find the addresses by streetname

EXERCISE: COLLECTIONS

- Write a program for having lunch with the whole group using the most approriate collections

- Everyone needs to stand in line to get lunch, add everyone to the queue and log the output of everyone getting lunch
- After lunch everyone piles their plates in the kitch, log every stacking action
- Every day someone will do the dishes, log whose dish is being washed and put into the the drying rack
- Everyday someone will dry the plates and make a new pile in the cupboard and log the order

ASSEMBLIES

- Unit of:
  - Deployment
  - Activation scoping
  - Security permission

- .exe and .dll files are possible extensions for assemblies

- Assemblies can be shared between two applications

- Only loaded into memory when they're used

- Assembly has assembly manifest file that lists all dependencies

CONSTANT

Compile time constants → const:

- Values that are set at compile time, cannot be changed, field can only be set when field is declared
- Keyword “const”
- The static modifier is not allowed in a constant declaration

Runtime constants → readonly:

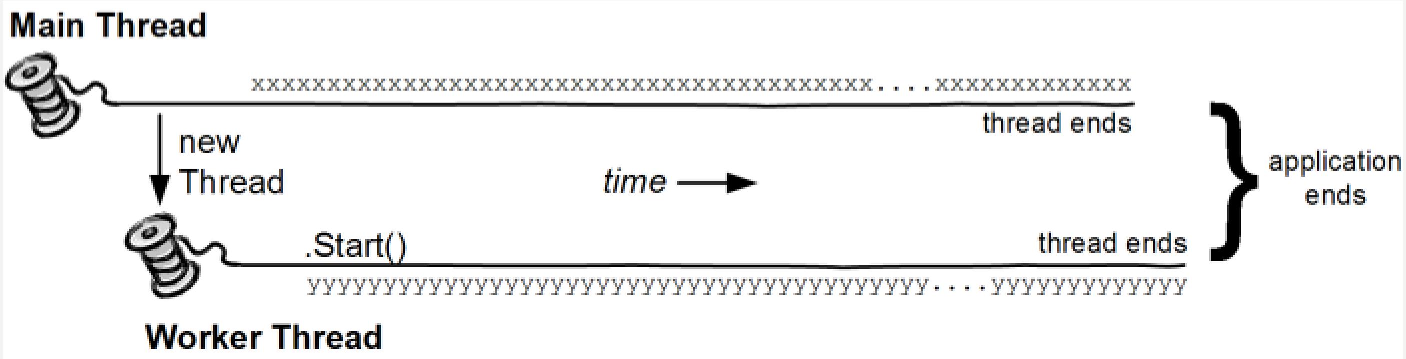
- Value can either be set in the declaration or in the constructor
- Readonly values cannot be changed after the constructor of the class they ‘re in is done

```
static class Constants
{
    public const double pi = 3.14159;
    public const int speedOfLight = 300000; // km per sec.
}

class Age
{
    readonly int year;

    Age(int year)
    {
        this.year = year;
    }//year can’t be changed after constructor
}
```

MULTITHREADING

- parallel execution of code
- A thread is an independent execution path, able to run simultaneously with other threads.
- Join and Sleep threads

```
using System.Threading;

Thread t = new Thread(WriteY);
// Kick off a new thread
t.Start();   // running WriteY()

// Simultaneously, do something on the main thread.
   for (int i = 0; i< 1000; i++) Console.Write("x");
 }
static void WriteY()
   {
       for (int i = 0; i < 1000; i++) Console.Write("y");
   }
```


Main Thread
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx....xxxxxxxxxxxxx
thread ends
new Thread
time
.Start()
thread ends
application ends
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy....yyyyyyyyyyyyy
Worker Thread

CONCURRENT COLLECTIONS

- In System.Collections.Concurrent namespace
- Thread-safe collections
- Adding and removing can be done safely, without worrying about synchronization
- Only use this when you are working with multiple threads

Name collection
ConcurrentDictionary<Tkey, Tvalue>
ConcurrentQueue<T>
ConcurrentStack<T>
ConcurrentBag<T>
BlockingCollection<T>
IProducerConsumerCollection<T>

EXAMPLE CONCURRENT COLLECTION

KAN NIET

```
 var list = new List<string>();
 list.Add("maaike");

foreach (string naam in list) {
    Console.WriteLine(naam);
    list.Add("nog een string");
}
```

KAN WEL

```
Var list = new BlockingCollection<string>();
            list.Add("maaike");

            foreach (string naam in list) {
                Console.WriteLine(naam);
                list.Add("nog een string");
            }
```

WHEN TO USE WHICH COLLECTION

- You want to create a shopping list
- You need to hold 4 numbers
- You want to store id’s and names
- You want to store elements that will be loaded in a truck and garantee you use the right order (LIFO) to take them out again
- You want to put people in a waiting room and deal with them on FIFO base

LINQ FOR COLLECTION ACCESS

- LINQ = language integrated query
- Can be used to access collections and data sources
- Easy ways for filtering, sorting, grouping of collections
- System.Linq namespace

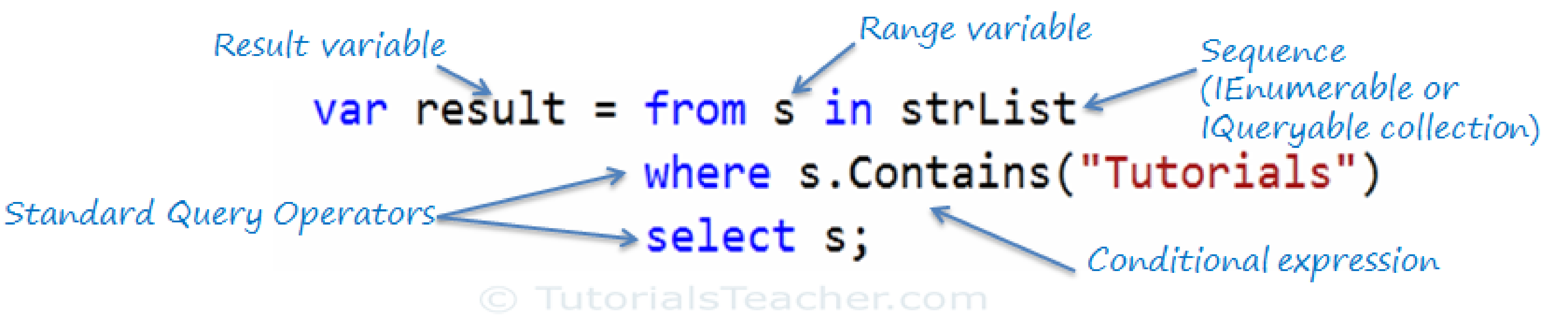
```
List<Element> elements = new List<Element>
  {
      { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
      { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
      { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
      { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
  };

  // LINQ
  var subset = from theElement in elements
               where theElement.AtomicNumber < 22
               orderby theElement.Name
               select theElement;
```

LINQ FOR COLLECTION ACCESS

- Results are returned as objects

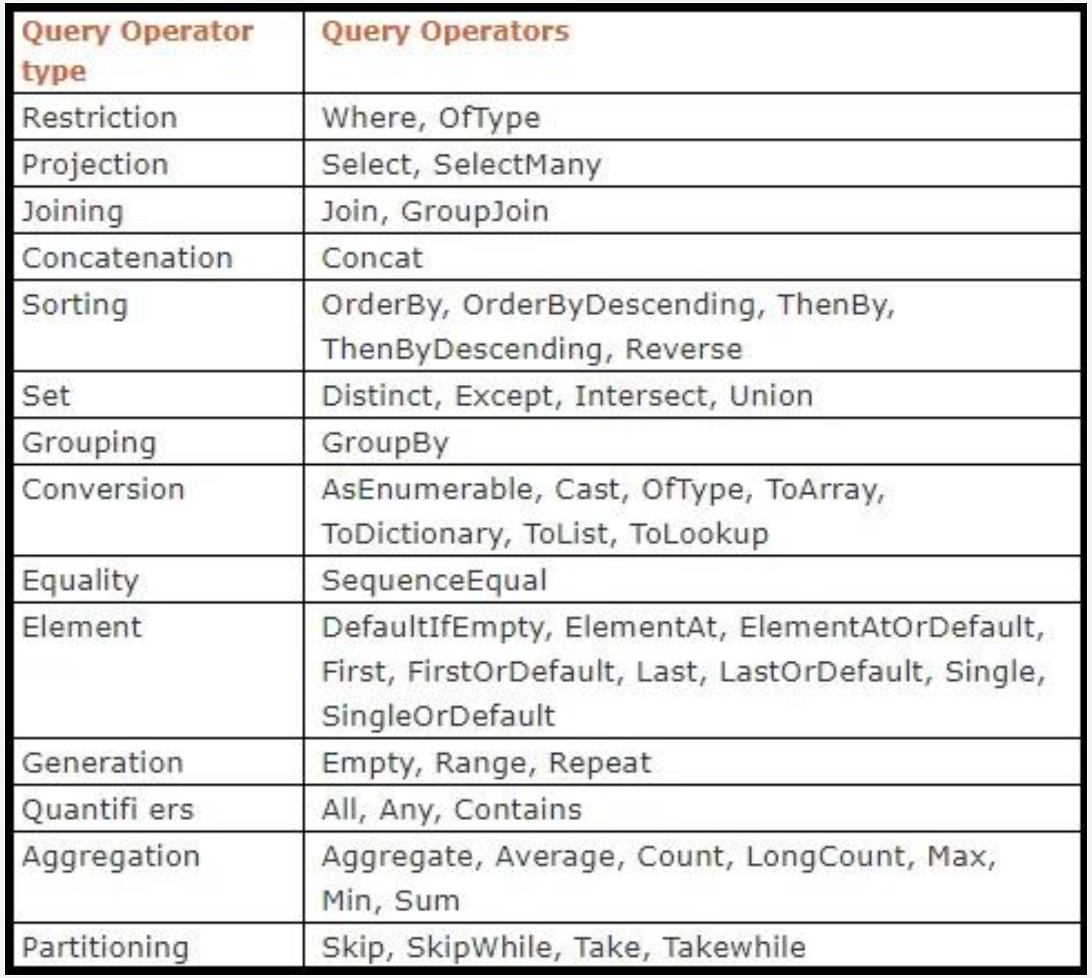
- Basic elements:
  - From
  - Where
  - Select

```
// Data source
string[] names = {"Bill", "Steve", "James", "Mohan" };

// LINQ Query
var myLinqQuery = from name in names
                  where name.Contains('a')
                  select name;

// Query execution
foreach(var name in myLinqQuery)
    Console.Write(name + " ");
```



```
var result = from s in strList
             where s.Contains("Tutorials")
             select s;
```

OTHER ELEMENTSS



Query Operator type	Query Operators
Restriction	Where, OfType
Projection	Select, SelectMany
Joining	Join, GroupJoin
Concatenation	Concat
Sorting	OrderBy, OrderByDescending, ThenBy, ThenByDescending, Reverse
Set	Distinct, Except, Intersect, Union
Grouping	GroupBy
Conversion	AsEnumerable, Cast, OfType, ToArray, ToDictionary, ToList, ToLookup
Equality	SequenceEqual
Element	DefaultIfEmpty, ElementAt, ElementAtOrDefault, First, FirstOrDefault, Last, LastOrDefault, Single, SingleOrDefault
Generation	Empty, Range, Repeat
Quantifiers	All, Any, Contains
Aggregation	Aggregate, Average, Count, LongCount, Max, Min, Sum
Partitioning	Skip, SkipWhile, Take, Takewhile

EXERCISE LINQ

- Create an array of numbers to 100
- Use LINQ to get the even numbers

- Create an array of names
- Use LINQ to get the names starting with an M or J

```
List<Element> elements = new List<Element>
  {
      { new Element() { Symbol="K", Name="Potassium", AtomicNumber=19}},
      { new Element() { Symbol="Ca", Name="Calcium", AtomicNumber=20}},
      { new Element() { Symbol="Sc", Name="Scandium", AtomicNumber=21}},
      { new Element() { Symbol="Ti", Name="Titanium", AtomicNumber=22}}
  };

  // LINQ
  var subset = from theElement in elements
               where theElement.AtomicNumber < 22
               orderby theElement.Name
               select theElement;
```

ORDERBY

- Ordering in a collection according to some key
- Default is ascending. **OrderByDescending** to reverse
- **ThenBy** and **ThenByDescending** specify subsequent ordering

EXERCISE LINQ

- Create a list of animals
- Use LINQ to get the animals older than 5
- Order them first by name
- Then by age

JOINING

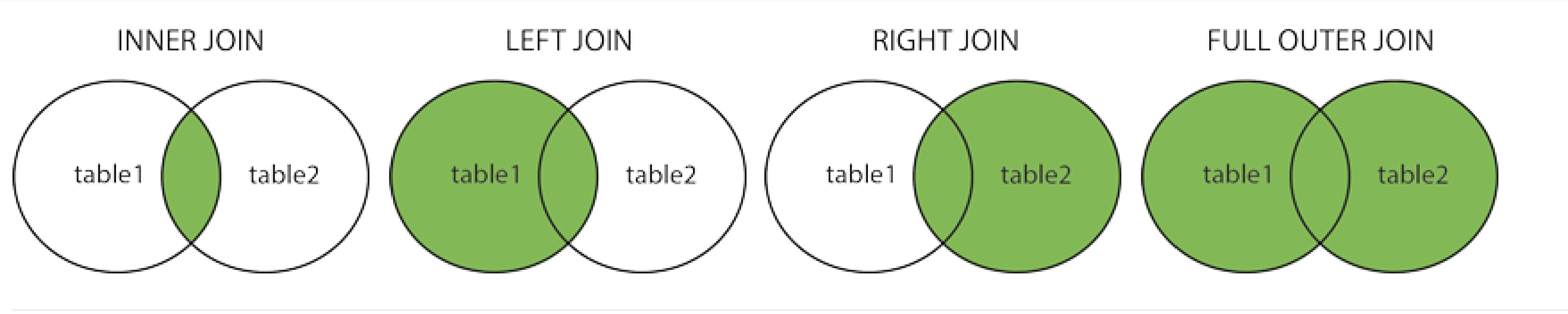
- Used to combine rows from two or more tables, based on a related column between them.

- similar to an inner join in SQL

- Example in SQL
  - SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
    FROM Orders
    INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
- Example in LINQ
  - From
    join … in …
    on … equals …

DIFFERENT TYPES OF JOIN

- (INNER) JOIN: Returns records that have matching values in both tables
- LEFT (OUTER) JOIN: Returns all records from the left table, and the matched records from the right table
- RIGHT (OUTER) JOIN: Returns all records from the right table, and the matched records from the left table
- FULL (OUTER) JOIN: Returns all records when there is a match in either left or right table



INNER JOIN EXAMPLE

```
var innerjoin =
from d in dogs
join owner in owners
on d.OwnerID equals owner.Id
select new
{
    OwnerName = owner.Name,
    DogName = dog.Name
};
```

GROUP JOIN

- Uses the into keyword to group

- Inner join that uses grouping

```
var groupjoin =
from owner in owners
orderby owner.ID
join d in dogs
on owner.ID equals animal.OwnerID
into ownerGroup
select new
{
        Owner = owner.Name,
        Animals = from owner2
                in ownerGroup
                orderby owner2.Name
                select owner2
};
```

EXERCISE LINQ

- Create a new class owner
- Give your dog an owner
- Make two lists, one with dogs, one with owners
- First create a query that holds ownername and dogname
- Bonus: Then make a query that groups dog names by owner name

SELECT

- Can also be used to create new collection

```
From d in dogs
Select new
{
    d.Name,
    d.Age
};
```

EXERCISE LINQ

- Create an array of numbers 0 to 25
- Use LINQ to create a new array that shows the number and its square

EXERCISE (DAG 3 ROUNDUP)

- Aks the user for numeric input 5 times and store this into one variable
- Go over the user input to determine whether it contains prime number
- Remove the none prime numbers from the list
- Print that you checked for prime numbers and show the remaining values

WRAP UP DAG 3

- Arrays
- Generics
- Indexers
- Collections
- Foreach
- Assemblies
- Constants
- LINQ

NOG VRAGEN?

Stel ze per mail:

Maaike.vanputten@brightboost.nl

Of WhatsApp:

0683982426