Для выбора элементов из некоторого набора по условию используется метод **Where**:

```
Where<TSource> (Func<TSource,bool> predicate)
```

Этот метод принимает делегат `Func<TSource,bool>`, который в качестве параметра принимает каждый элемент последовательности и возвращает значение bool. Если элемент соответствует некоторому условию, то возвращается true, и тогда этот элемент передаетсяв коллекцию, которая возвращается из метода Where.

Например, выберем все строки, длина которых равна 3:

```
string[] people = { "Tom", "Alice", "Bob", "Sam", "Tim", "Tomas", "Bill" };

var selectedPeople = people.Where(p => p.Length == 3); // "Tom", "Bob", "Sam", "Tim"

foreach (string person in selectedPeople)
    Console.WriteLine(person);
```

Если выражение в методе Where для определенного элемента будет равно true (в данном случае выражение `p.Length == 3`), то данный элемент попадает в результирующую выборку.

Аналогичный запрос с помощью операторов LINQ:

```
string[] people = { "Tom", "Alice", "Bob", "Sam", "Tim", "Tomas", "Bill" };

var selectedPeople = from p in people
                     where p.Length == 3
                     select p;
```

Другой пример - выберем все четные элементы, которые больше 10.

Фильтрация с помощью операторов LINQ:

```
int[] numbers = { 1, 2, 3, 4, 10, 34, 55, 66, 77, 88 };
// методы расширения
var evens1 = numbers.Where(i => i % 2 == 0 && i > 10);
// операторы запросов
var evens2 = from i in numbers
             where i%2==0 && i>10
             select i;
```

Выборка сложных объектов

Допустим, у нас есть класс пользователя:

```
record class Person(string Name, int Age, List<string> Languages);
```

Свойство Name представляет имя, свойство Age - возраст пользователя, а список Languages - список языков, которыми владеет пользователь.

Создадим набор пользователей и выберем из них тех, которым больше 25 лет:

```
var people = new List<Person>
{
    new Person ("Tom", 23, new List<string> {"english", "german"}),
    new Person ("Bob", 27, new List<string> {"english", "french" }),
    new Person ("Sam", 29, new List<string>  { "english", "spanish" }),
    new Person ("Alice", 24, new List<string> {"spanish", "german" })
};

var selectedPeople = from p in people
                     where p.Age > 25
                     select p;

foreach (Person person in selectedPeople)
    Console.WriteLine($"{person.Name} - {person.Age}");
```

Консольный вывод:

```
Bob - 27
Sam - 29
```

Аналогичный запрос с помощью метода расширения `Where`:

```
var selectedPeople = people.Where(p=> p.Age > 25);
```

Сложные фильтры

Теперь рассмотрим более сложные фильтры. Например, в классе пользователя есть список языков, которыми владеет пользователь. Что если нам надо отфильтровать пользователей по языку:

```
var selectedPeople = from person in people
                     from lang in person.Languages
                     where person.Age < 28
                     where lang == "english"
                     select person;
```

Результат:

```
Tom - 23
Bob - 27
```

Для создания аналогичного запроса с помощью методов расширения применяется метод **SelectMany**:

```
var selectedPeople = people.SelectMany(u => u.Languages,
                           (u, l) => new { Person = u, Lang = l })
                          .Where(u => u.Lang == "english" && u.Person.Age < 28)
                          .Select(u=>u.Person);
```

Метод `SelectMany()` в качестве первого параметра принимает последовательность, которую надо проецировать, а в качестве второго параметра - функцию преобразования, которая применяется к каждому элементу. На выходе она возвращает 8 пар "пользователь - язык" (new { Person = u, Lang = l }), к которым потом применяется фильтр с помощью Where.

Фильтрация по типу данных

Дополнительный метод расширения - **OfType()** позволяет отфильтровать данные коллекции по определенному типу:

```
var people= new List<Person>
{
    new Student("Tom"),
    new Person("Sam"),
    new Student("Bob"),
    new Employee("Mike")
};

var students = people.OfType<Student>();

foreach (var student in students)
    Console.WriteLine(student.Name);


record class Person(string Name);
record class Student(string Name): Person(Name);
record class Employee(string Name) : Person(Name);
```

В данном случае список people содержит объекты трех типов - класса Person и производных типов Student и Employee. И в примере производится фильтрация данных типа Student - для этого метод `OfType()` типизируется типом Student. Консольный вывод:

```
Tom
Bob
```