**Records**

***Records*** представляют новый ссылочный тип, который появился в C#9. Ключевая особенность records состоит в том, что они могут представлять неизменяемый (immutable) тип, который по умолчанию обладает рядом дополнительных возможностей по сравнению с классами и структурами. Зачем нам нужны неизменяемые типы? Такие типы более безопасны в тех ситуациях, когда нам надо гарантировать, что данные объекта не будут изменяться. В .NET в принципе уже есть неизменяемые типы, например, String.

Для определения records используется ключевое слово record. Если определяется класс record, то ключевое слово class можно неиспользовать при определении типа:

public record Person

{

public string Name { get; set; }

public Person(string name) => Name = name;

}

или так

public record class Person

{

public string Name { get; set; }

public Person(string name) => Name = name;

}

При определении структуры ***record*** при объявлении типа надо использовать ключевое слово ***struct***:

public record struct Person

{

public string Name { get; set; }

public Person(string name) => Name = name;

}

Хотя типы ***record*** предназначены для создания неизменяемых типов, однако одно только применение ключевого слова record не гарантирует неизменяемость объектов record. Они являются неизменяемыми (immutable) только при определенных условиях. Например, мы можем написать так:

var person = new Person("Tom");

person.Name = "Bob";

Console.WriteLine(person.Name); // Bob - данные изменились

public record Person

{

public string Name { get; set; }

public Person(string name) => Name = name;

}

При выполнении этого кода не возникнет никакой ошибки, мы спокойно сможем изменять значения свойств объекта Person. Чтобы сделать его действительно неизменяемым, для свойств вместо обычных сеттеров надо использовать модификатор ***init***.

var person = new Person("Tom");

person.Name = "Bob"; // ! ошибка - свойство изменить нельзя

public record Person

{

public string Name { get; init; }

public Person(string name) => Name = name;

}

В данном случае мы получим ошибку при попытке изменить значение свойств объекта Person.

Во многим ***records*** похожи на обычные классы и структуры, например, они могут абстрактными, их также можно наследовать либо запрещать наследование с помощью оператора sealed. Тем не менее есть и ряд отличий. Рассмотрим некоторые основные отличия records от стандартных классов и структур.

**Сравнение на равенство**

При определении record компилятор генерирует метод **Equals()** для сравнения с другим объектом. При этом сравнение двух **records** производится на основе их значений. Например, рассмотрим следующий пример:

var person1 = new Person("Tom");

var person2 = new Person("Tom");

Console.WriteLine(person1.Equals(person2)); // true

var user1 = new User("Tom");

var user2 = new User("Tom");

Console.WriteLine(user1.Equals(user2)); // false

public record Person

{

public string Name { get; init; }

public Person(string name) => Name = name;

}

public class User

{

public string Name { get; init; }

public User(string name) => Name = name;

}

В данном случае при сравнении двух объектов record Person мы увидим, что они равны, так как их значения (значения свойств Name) равны. Однако в случае с объектами класса User, которые имеют те же одинаковые значения мы увидим, что они не равны. Так как сравнение records производится по значению.

Кроме того, для **record** уже по умолчанию реализованы операторы ***==*** и ***!=***, которые также сравнивают две record по значению:

var person1 = new Person("Tom");

var person2 = new Person("Tom");

Console.WriteLine(person1 == person2); // true

var user1 = new User("Tom");

var user2 = new User("Tom");

Console.WriteLine(user1 == user2); // false

**Оператор with**

В отличие от классов *records* поддерживают инициализацию с помощью оператора with. Он позволяет создать одну record на основе другой **record**:

var tom = new Person("Tom", 37);

var sam = tom with { Name = "Sam" };

Console.WriteLine($"{sam.Name} - {sam.Age}"); // Sam - 37

public record Person

{

public string Name { get; init; }

public int Age { get; init; }

public Person(string name, int age)

{

Name = name; Age = age;

}

}

После **record**, значения которой мы хотим скопировать, указывается оператор **with**, после которого в фигурных скобках указываются значения для тех свойств, которые мы хотим изменить. Так, в данном случае переменная sam получает для свойства Age значение из tom, а свойство Name изменяется.

Эта возможность может быть особенно актуальна, если в **record**, которую мы хотим скопировать, множество свойств, из которых мы хотим поменять одно-два.

Если надо скопировать значения всех свойств, то можно оставить пустые фигурные скобки:

var person1 = new Person("Tom", 37);

var person2 = person1 with { };

**Позиционные records**

Records могут принимать данные для свойств через конструктор, и в этом случае мы можем сократить их определение. Например, пусть у нас есть следующая record Person:

public record Person

{

public string Name { get; init; }

public int Age { get; init; }

public Person(string name, int age)

{

Name = name; Age = age;

}

public void Deconstruct(out string name, out int age) => (name, age) = (Name, Age);

}

Кроме конструктора здесь реализован деконструктор, который позволяет разложить объект Person на кортеж значений. И мы могли бы применить ее, например, следующим образом:

var person = new Person ("Tom", 37);

Console.WriteLine(person.Name); // Tom

var (personName, personAge) = person;

Console.WriteLine(personAge); // 37

Console.WriteLine(personName); // Tom

Выше определенную record Person можно сократить до позиционной record:

public record Person(string Name, int Age);

Это все определение типа. То есть мы говорим, что для типа Person будет создаваться конструктор, который принимает два параметра и присваивает их значения соответственно свойствам Name и Age, и что также автоматически будет создаваться деконструктор. Ее использование будет аналогично:

var person = new Person("Tom", 37);

Console.WriteLine(person); // Tom

var (personName, personAge) = person;

Console.WriteLine(personAge); // 37

Console.WriteLine(personName); // Tom

public record Person(string Name, int Age);

При необходимости также можно совмещать стандартное определение свойств и определение свойств через конструктор:

var person = new Person("Tom", 37) { Company = "Google"};

Console.WriteLine(person.Company); // Google

person.Company = "Microsoft";

Console.WriteLine(person.Company); // Microsoft

public record Person(string Name, int Age)

{

public string Company { get; set; } = "";

}

**Позиционные структуры для чтения**

Следует отметить различие между позиционными классами и структурами record. Свойства класса record, которые устанавливаются через параметры конструктора, по умолчанию будут иметь модификатор init. То есть после установки их значений через конструктор, мы больше не сможем их изменить:

var person = new Person("Tom", 37);

person.Name = "Bob"; // ! Ошибка - значение нельзя изменить

public record Person(string Name, int Age);

Стоит отметить, что это относится только к тем свойствам, которые устанавливаются через конструктор.

Однако для позиционных структур record свойства будут иметь стандартные сеттеры, которые позволят изменять значения свойств:

var person = new Person("Tom", 37);

person.Name = "Bob";

Console.WriteLine(person.Name); // Bob - значение изменилось

// структура record

public record struct Person(string Name, int Age);

Чтобы для подобных свойств структуры record использовался модификатор init вместо обычных сеттеров, такую структуру надо определить с ключевым словом readonly:

var person = new Person("Tom", 37);

person.Name = "Bob"; // ! Ошибка - значение свойства нельзя изменить

// структура record доступна только для чтения

public readonly record struct Person(string Name, int Age);

**ToString**

Небольшим преимуществом типов record также является то, что для них уже по умолчанию реализован метод **ToString()**, который выводит состояние объекта в отформатированном виде:

var person = new Person("Tom", 37);

Console.WriteLine(person); // Person {Name = Tom, Age = 37}

public record Person(string Name, int Age);

**Наследование**

Как и обычные классы record-классы могут наследоваться:

var tom = new Person("Tom", 37);

var bob = new Employee("Bob", 41, "Microsoft");

Console.WriteLine(tom); // Person {Name = Tom, Age = 37}

Console.WriteLine(bob); // Person {Name = Bob, Age = 41, Company = Microsoft}

public record Person(string Name, int Age);

public record Employee(string Name, int Age, string Company) : Person(Name, Age);

В данном случае класс record Employee наследуется от Person.