**Module: Object-Oriented Programming (OOP).**

**Lesson: №10.**

**Theme: OOPs concept.**

**Lesson Content:**

* **OOPs concept.**
  + **What is the Namespace.**
  + **Records**
  + **Struct.**
    - **Ref structures.**
    - **Readonly structures.**

**Sub-Theme: What is Namespace?**

The namespace keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.



File scoped namespace declarations enable you to declare that all types in a file are in a single namespace. File scoped namespace declarations are available with C# 10. The following example is similar to the previous example, but uses a file scoped namespace declaration:



The preceding example doesn't include a nested namespace. File scoped namespaces can't include additional namespace declarations. You cannot declare a nested namespace or a second file-scoped namespace:



Within a namespace, you can declare zero or more of the following types:

* class;
* interface;
* struct;
* enum;
* delegate;
* nested namespaces can be declared except in file scoped namespace declarations;

The compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. It contains declarations not included in a declared namespace. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access. For a discussion of the access modifiers you can assign to elements in a namespace.

It's possible to define a namespace in two or more declarations. For example, the following example defines two classes as part of the MyCompany namespace:



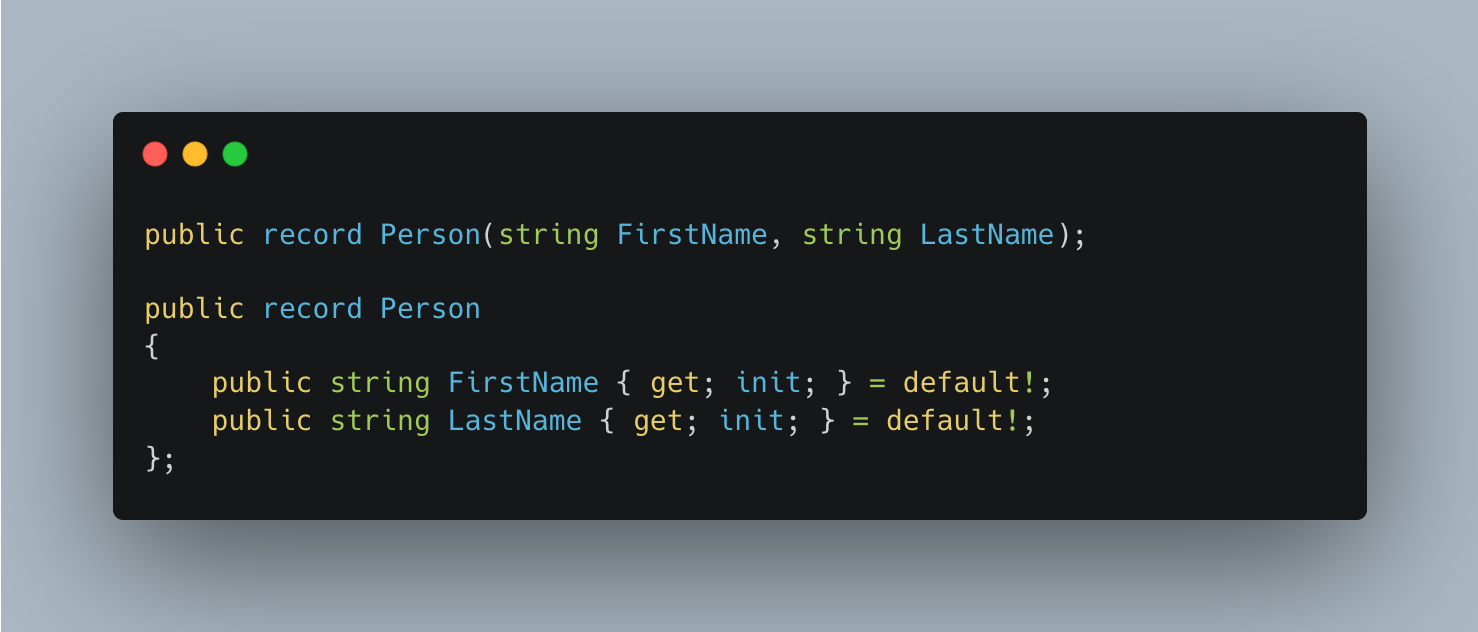
The following example shows how to call a static method in a nested namespace.



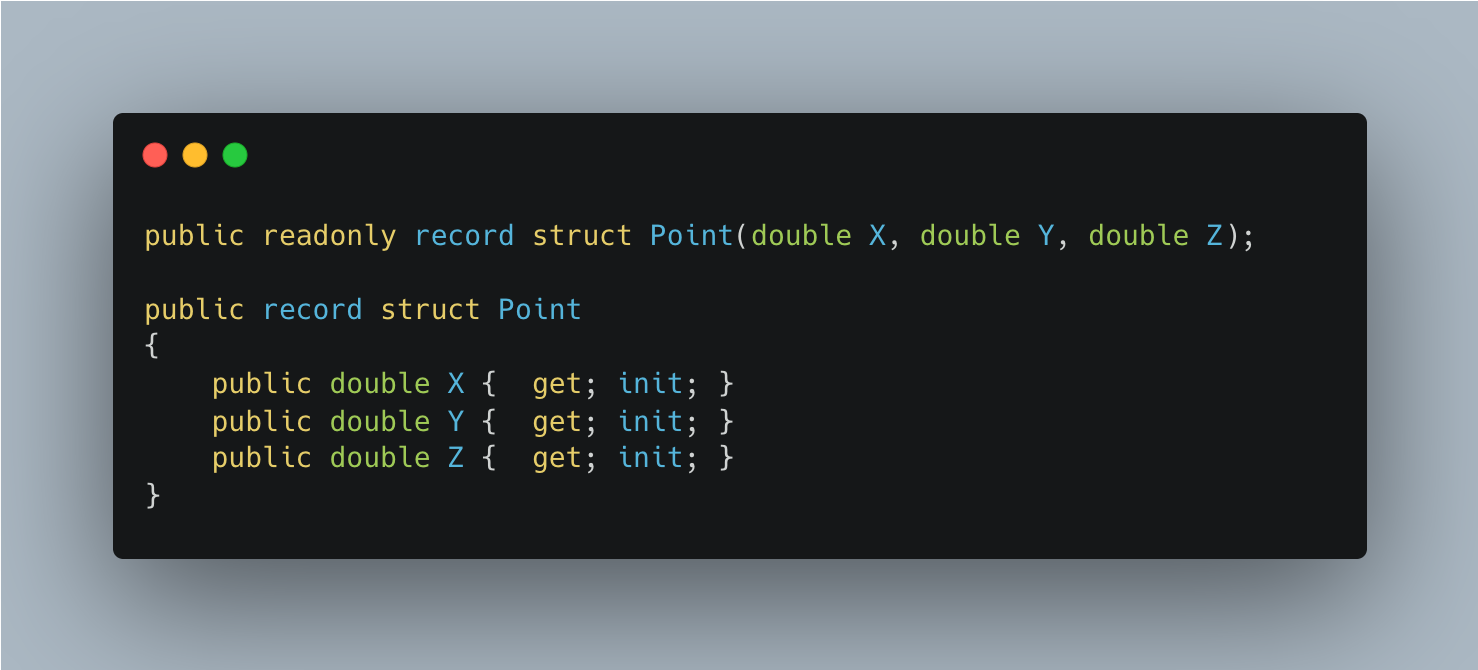
**Sub-Theme: Records.**

Beginning with C# 9, you use the record keyword to define a reference type that provides built-in functionality for encapsulating data. C# 10 allows the record class syntax as a synonym to clarify a reference type, and record struct to define a value type with similar functionality. You can create record types with immutable properties by using positional parameters or standard property syntax.

The following two examples demonstrate record (or record class) reference types:



The following two examples demonstrate record struct value types:



**Immutability** can be useful when you need a data-centric type to be thread-safe or you're depending on a hash code remaining the same in a hash table.

Immutability isn't appropriate for all data scenarios, however. Entity Framework Core, for example, doesn't support updating with immutable entity types.

Init-only properties, whether created from positional parameters (record class, and readonly record struct) or by specifying init accessors, have shallow immutability. After initialization, you can't change the value of value-type properties or the reference of reference-type properties. However, the data that a reference-type property refers to can be changed. The following example shows that the content of a reference-type immutable property (an array in this case) is mutable:



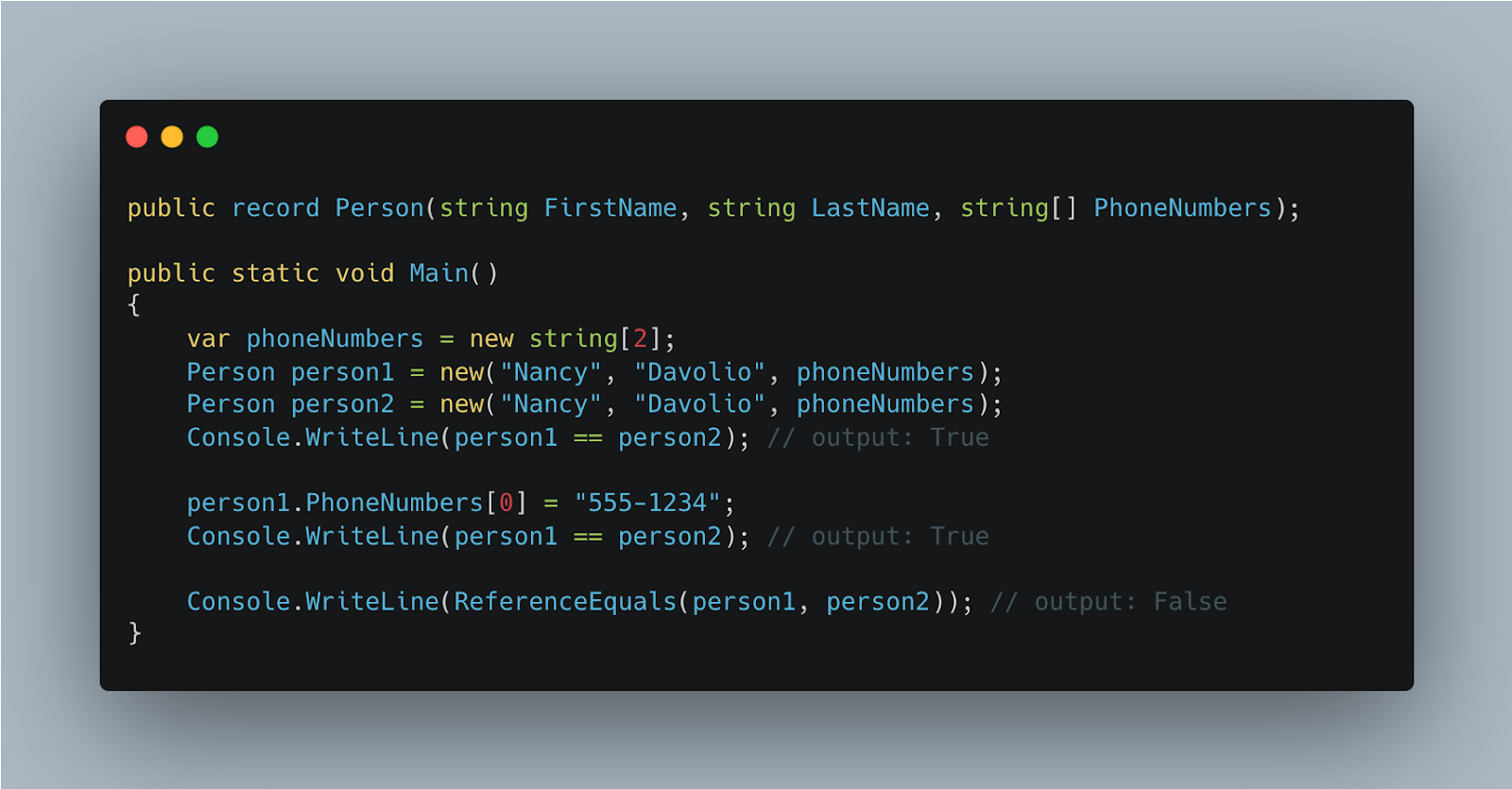
The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state. Unless specified, the synthesized methods are generated for record, record struct, and readonly record struct declarations.

if you don't override or replace **equality** methods, the type you declare governs how equality is defined:

* For class types, two objects are equal if they refer to the same object in memory.
* For struct types, two objects are equal if they are of the same type and store the same values.
* For record types, including record struct and readonly record struct, two objects are equal if they are of the same type and store the same values.

The definition of equality for a record struct is the same as for a struct. The difference is that for a struct, the implementation is in ValueType.Equals(Object) and relies on reflection. For records, the implementation is compiler synthesized and uses the declared data members.

Reference equality is required for some data models. For example, Entity Framework Core depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, records and record structs aren't appropriate for use as entity types in Entity Framework Core. The following example illustrates value equality of record types:



If you need to copy an instance with some **modifications**, you can use a with expression to achieve nondestructive mutation. A with expression makes a new record instance that is a copy of an existing record instance, with specified properties and fields modified. You use object initializer syntax to specify the values to be changed, as shown in the following example:



The with expression can set positional properties or properties created by using standard property syntax. Non-positional properties must have an init or set accessor to be changed in a with expression.

The result of a with expression is a shallow copy, which means that for a reference property, only the reference to an instance is copied. Both the original record and the copy end up with a reference to the same instance.

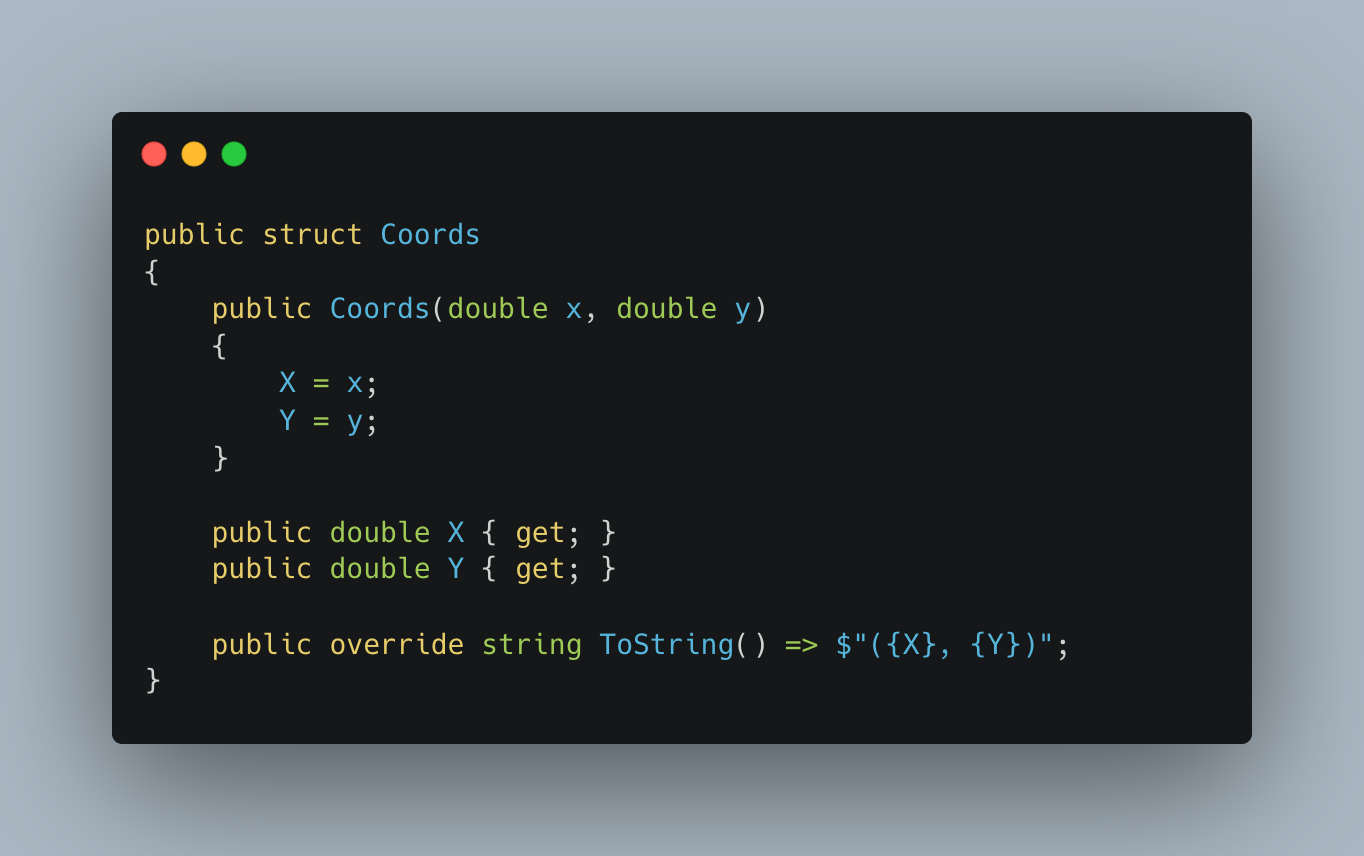
To implement this feature for record class types, the compiler synthesizes a clone method and a copy constructor. The virtual clone method returns a new record initialized by the copy constructor. When you use a with expression, the compiler creates code that calls the clone method and then sets the properties that are specified in the with expression.

If you need different copying behavior, you can write your own copy constructor in a record class. If you do that, the compiler won't synthesize one. Make your constructor private if the record is sealed, otherwise make it protected. The compiler doesn't synthesize a copy constructor for record struct types. You can write one, but the compiler won't generate calls to it for with expressions. The values of the record struct are copied on assignment.

You can't override the clone method, and you can't create a member named Clone in any record type. The actual name of the clone method is compiler-generated.

**Sub-Theme: Structure.**

A structure type (or struct type) is a value type that can encapsulate data and related functionality.



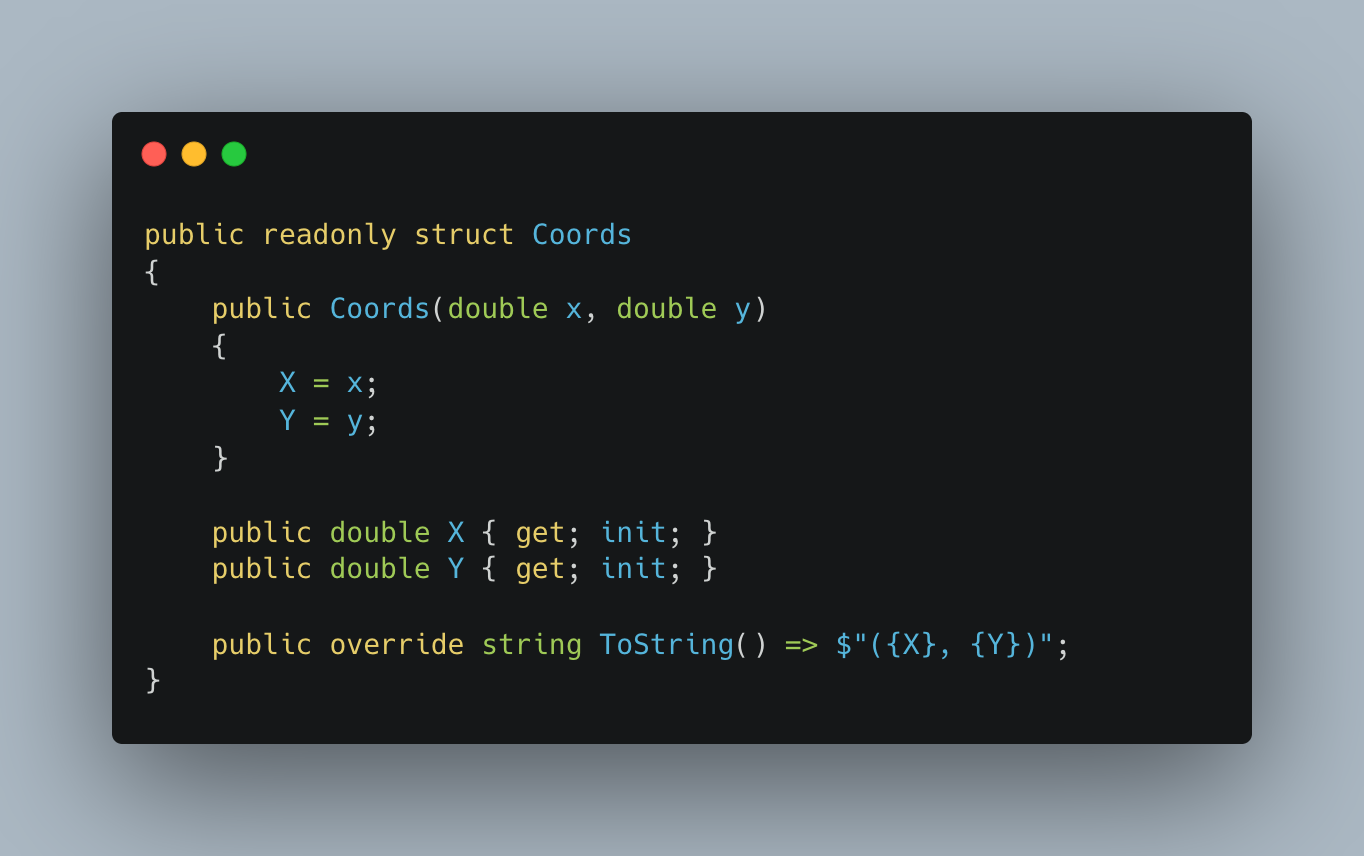
Structure types have value semantics. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. For structure-type variables, an instance of the type is copied. For more information, see Value types.

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both integer and real), a Boolean value, a Unicode character, a time instance. If you're focused on the behavior of a type, consider defining a class. Class types have reference semantics. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

Because structure types have value semantics, we recommend you define immutable structure types.

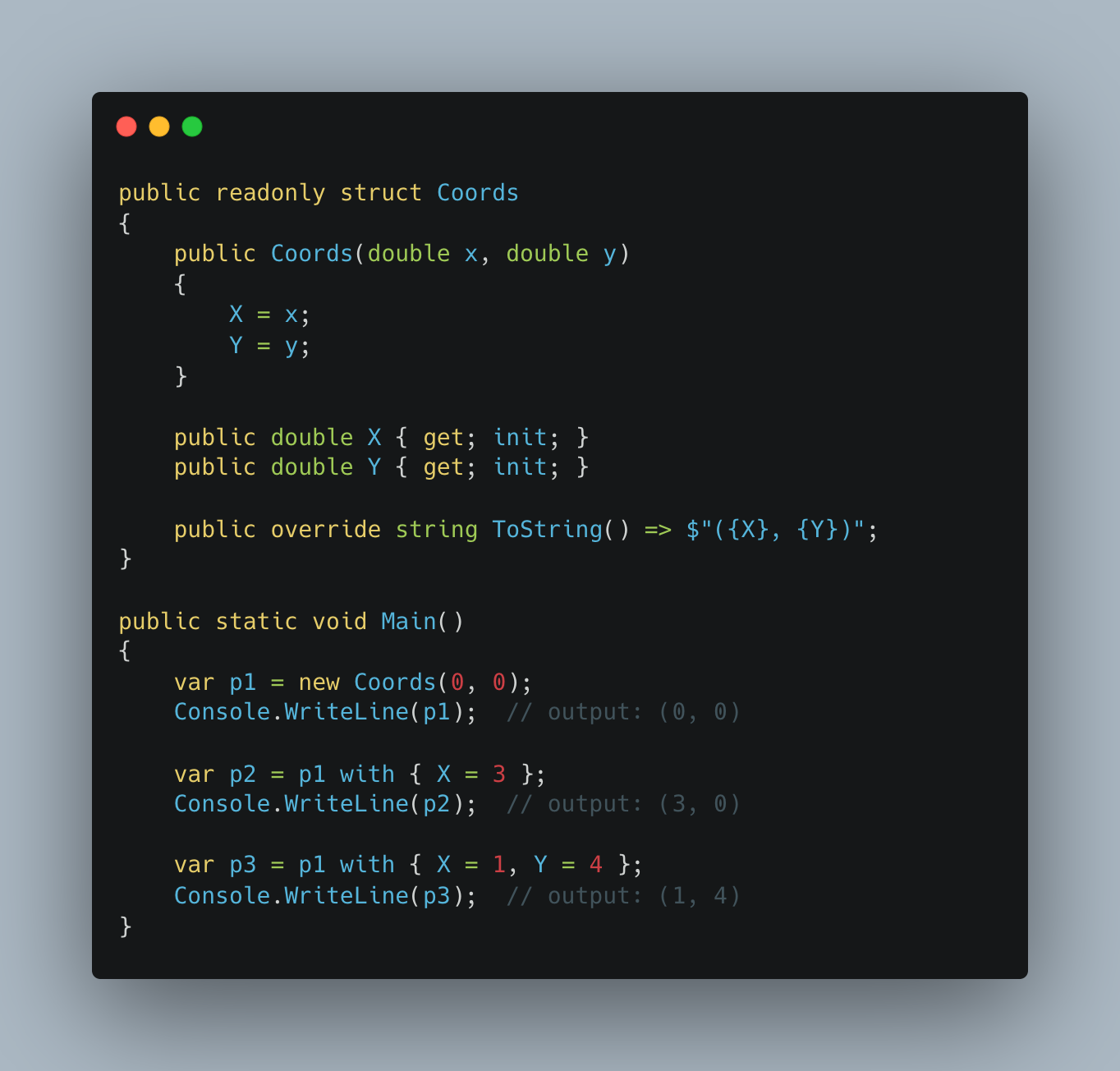
You use the readonly modifier to declare that a structure type is immutable. All data members of a readonly struct must be read-only as follows:

* Any field declaration must have the readonly modifier
* Any property, including auto-implemented ones, must be read-only. In C# 9.0 and later, a property may have an init accessor.



That guarantees that no member of a readonly struct modifies the state of the struct. That means that other instance members except constructors are implicitly readonly.

Beginning with C# 10, you can use the with expression to produce a copy of a structure-type instance with the specified properties and fields modified. You use object initializer syntax to specify what members to modify and their new values, as the following example shows:



When you pass a structure-type variable to a method as an argument or return a structure-type value from a method, the whole instance of a structure type is copied. Pass by value can affect the performance of your code in high-performance scenarios that involve large structure types. You can avoid value copying by passing a structure-type variable by reference. Use the ref, out, or in method parameter modifiers to indicate that an argument must be passed by reference. Use ref returns to return a method result by reference.

You can use the ref modifier in the declaration of a structure type. Instances of a ref struct type are allocated on the stack and can't escape to the managed heap. To ensure that, the compiler limits the usage of ref struct types as follows:

* A ref struct can't be the element type of an array.
* A ref struct can't be a declared type of a field of a class or a non-ref struct.
* A ref struct can't implement interfaces. A ref struct can't be boxed to System.ValueType or System.Object.
* A ref struct can't be a type argument.
* A ref struct variable can't be captured by a lambda expression or a local function.
* A ref struct variable can't be used in an async method. However, you can use ref struct variables in synchronous methods, for example, in methods that return Task or Task<TResult>.
* A ref struct variable can't be used in iterators.

**Homework:**

1. Define the term "namespace" in the context of the C# language. Explain its purpose and how it helps organise code elements.
2. Describe the concept of file-copy namespace declarations in C# 10. How do they differ from regular namespaces?
3. Explain the meaning of immutability in record types. Give examples of scenarios where immutability is critical.
4. Give the definition and distinction between the concepts of "record class" and "record structure" in C# 10. Provide code examples for both types.
5. What are the types of structures in C#? Explain the fundamental characteristics that distinguish them from class types.
6. Discuss the concept of passing by value and passing by reference when working with struct types. How can you avoid copying values for large structural types?