2
VALUE AND REFERENCE TYPES

We can create new types in C# in several ways, and we need to consider the individual characteristics of each approach to determine which best suits our goals. In particular, knowing how value types differ from reference types helps us choose the right way forward, because these differences have significant and sometimes unconsidered implications when we're defining our own types. Certain trade-offs will affect how we design our type and what we can use it for. In this chapter, we'll investigate those differences and what they mean for our programs.

We'll explore the following:

- What choices we have when creating our own types
- Why C# has both reference types and value types

- How choosing one or the other affects construction, null checking, and other type behavior
- Why value type is not the same as value semantics
- Where different types are stored in memory and how that affects an object's lifetime

User-Defined Types

Most modern programming languages allow you to create custom types. The basic principles of user-defined types in C# will be familiar to programmers of many other languages, but some of the details are different. Therefore, in this section we'll examine the four kinds of user-defined types: structs, classes, and the newer records and record structs (introduced in C# v9.0 and v10.0, respectively).

It's important to recognize that the behavior of these types relies heavily on whether they are reference types or value types. Let's look briefly at each kind of user-defined type with these differences in mind.

Structs and Classes

Listing 2-1 defines a simple `struct` to represent colors.

```
public readonly struct Color
{
    public Color(int r, int g, int b)
        => (Red, Green, Blue) = (r, g, b);

    public int Red { get; }
    public int Green { get; }
    public int Blue { get; }
}
```

Listing 2-1: Defining a simple struct

The `Color` struct is marked `readonly` to indicate that instances of `Color` are immutable—that is, they never change their value. Correspondingly, none of the three properties (`Red`, `Green`, and `Blue`) has a `set` accessor, so their values can't be changed after they've been given initial values using the constructor.

The constructor in this example uses the expression body syntax (`=>`), which you saw in Chapter 1, instead of a body enclosed between braces `{...}`. We make the expression body a single-line statement by using *tuple assignment*, which assigns the tuple of three parameter values `r`, `g`, and `b` to the tuple of three properties. The compiler translates this syntax into an efficient assignment from the parameter values directly to the respective backing fields for the `Red`, `Green`, and `Blue` properties.

The `readonly` keyword in the struct's definition is not mandatory but reinforces that instances of `Color` are immutable. Immutable value types make our code easier to comprehend and may allow some optimizations by the compiler.

By contrast, if we define a `Color` class instead of a struct, we can't use the `readonly` keyword in its definition, although we can make it immutable by not providing `set` accessors for the properties. The only other difference in defining `Color` as a class is the use of the `class` keyword in the definition:

```
public class Color
{
--snip--
```

The definition of `Color` is otherwise identical to that in Listing 2-1.

The principal difference between these two types is that a class is a reference type, and a struct is a value type. Before we analyze the implications of this difference, let's look at record and record struct types.

Records and Record Structs

As of C# v9.0, we can define a record type with the `record` keyword. Records introduce a new syntax for compactly defining a type. Listing 2-2 creates a record type named `Color`.

```
public record Color(int Red, int Green, int Blue);
```

Listing 2-2: Defining a record

This example shows a *positional* record; the `Color` type has no body, but the type definition has its own positional parameters that are used by the compiler to generate a complete type. Behind the scenes, the compiler translates the record into a class definition, meaning that records are reference types. The compiler also translates the parameter names `Red`, `Green`, and `Blue` into public properties of the same name, along with a public constructor with matching parameters to initialize the property values. The positional parameters are also used by the compiler to generate other methods, including `Equals`, `GetHashCode`, and `ToString`, which are overrides of their counterparts in the `object` base class.

Listing 2-3 creates a new instance of the `Color` record and uses its properties exactly as if it were a class or a struct.

```
var tomato = new Color(Red: 255, Green: 99, Blue: 71);

Assert.That(tomato.Red, Is.EqualTo(255));
Assert.That(tomato.Green, Is.EqualTo(99));
Assert.That(tomato.Blue, Is.EqualTo(71));
```

Listing 2-3: Creating an instance of `Color`

Here, we use named arguments when constructing the `tomato` variable of type `Color` to emphasize the names given by the compiler to the constructor parameters. Note that the property names used in the assertions are identical to the names used in the constructor, and that both match the names used in the record definition.

NOTE *One important difference between records and structs or classes relates to handling equality comparisons between two instances, a topic we'll examine in more detail in "Identity Equality vs. Value Equality" on page 47.*

Very closely related to records are record structs, introduced in C# v10.0. In contrast to records, which are compiled as classes, record structs are translated by the compiler into struct definitions, making them value types. Otherwise, they're the same as records. Record structs are denoted by the `record struct` keywords, as shown here:

```
public readonly record struct Color(int Red, int Green, int Blue);
```

This record struct, much like the struct in Listing 2-1, is marked `readonly`. If we left out the `readonly` keyword, the properties generated by the compiler would be read-write properties, with both `get` and `set` accessors. Using the `readonly` keyword makes `Color` an immutable record struct.

Inheritance

One common way of representing relationships between classes and between records is to use *inheritance*, or deriving one type from another. However, we can't apply inheritance to structs or record structs; it's available only to reference types.

Another restriction of inheritance is that a record can inherit from another record but not explicitly from a class. Similarly, classes can't inherit from records. In every other respect, records follow the same rules and have the same characteristics as classes as far as inheritance is concerned. Classes and records can define virtual methods and properties, allowing a more derived type to provide its own behavior by overriding the method or property, and we can choose to ignore, override, or hide any virtual methods in a derived type.

In contrast, structs and record structs are implicitly *sealed*, meaning that inheriting from them is prohibited. If we attempt to derive from a struct or record struct, we get a compile-time error. Structs and record structs can't inherit from another user-defined type either.

Another restriction for a class or record is that it can inherit from only one base type. Any attempt at multiple inheritance results in a compiler error. If no base type is explicitly specified, `object` becomes the implied base class. As you'll see in "The Common Type System" on page 45, every type ultimately inherits from `object`, either directly or indirectly. For example, the `Command` class in Listing 2-4 implicitly derives from `object`, while the `DummyCommand` class derives explicitly from `Command`, implicitly inheriting from `object` via the `Command` base class.

```
public class Command
{
    public virtual IEnumerable<Result> RunQuery(string query)
    {
        using var transaction = connection.BeginTransaction();
        return connection.Execute(transaction, query);
    }

    private readonly DatabaseConnection connection;
}

public class DummyCommand : Command
{
    public override IEnumerable<Result> RunQuery(string query)
    {
        return new List<Result>();
    }
}
```

Listing 2-4: Inheritance syntax

This `Command` base class defines a virtual `RunQuery` method, which is overridden in the derived `DummyCommand` class to alter the method's behavior. A stub implementation like `DummyCommand` might be used during testing to avoid having the test code depend on the underlying data store's contents.

Any type may *implement* multiple interfaces, but it's important to understand that inheritance is quite different from interface implementation. When we implement an interface, the implementing method is, by default, *not* virtual. A class or record implementing a method from an interface can choose to make its implementation of the method virtual, but a struct or record struct cannot.

We can explicitly designate any member of a class or record as `protected`, as opposed to `public`, `private`, or `internal`. A protected member is accessible within the class declaring it and to any types that inherit from that class, but it's not visible to any other code. Since value types are sealed, it makes no sense for them to have virtual or protected members. If we try to make a method virtual in a value type definition or to define any protected fields, properties, or methods, we're rewarded with a compiler error.

We can choose to declare a class or record type as `sealed` so that it can't be used for further inheritance. Sealing a class does not affect what *it* can inherit, only what can inherit from it. It's common to seal classes that have value-like characteristics, such as `string`, or when we wish to restrict a class's behavior to that defined in our own implementation. If a class is intended to be immutable, whether or not it's intended to have value-like characteristics, sealing it ensures that its immutability can't be subverted by a mutable derived class.

Records are specifically intended to be value-like types and have value-like behavior defined for them by the compiler. This means we should seal record types unless we have a compelling reason not to do so. We'll look in detail at the meaning of *value-like* and why such types should be sealed in Chapters 6 and 7.

ABSTRACT BASE TYPES

An *abstract* type is one that can be used only as a base type for inheritance; it can't be instantiated directly with `new`. One implication is that while classes and records can be abstract, structs and record structs can't. It would make no sense: we can't inherit from a value type, so it could never be instantiated.

In an abstract type, we can designate methods and properties as `abstract`, meaning they have no implementation. Their purpose is simply to define the operations that a concrete type must support. An abstract method or property is implicitly virtual, but providing an implementation for one prompts a compiler error. Abstract types don't have to define any abstract members, but only abstract classes or records can have abstract methods and properties. Any abstract methods or properties remain abstract unless they're explicitly overridden in a derived class. Providing an implementation for an abstract method in a derived type makes that method concrete.

We can inherit one abstract type from another and choose to either provide implementations for the base type's abstract methods or leave them as abstract. We can only directly create an instance of a class or record that is fully concrete; that is, any and all abstract methods have been overridden.

If we inherit from an abstract class, we can't then inherit from any other class because that would be a form of multiple inheritance, which is prohibited.

It can be tempting to think of C# interfaces and their members as being abstract (especially for users familiar with C++, where interfaces are commonly implemented as classes with all pure-virtual methods), but that's not the case. An interface contains only signatures of methods and properties; they are neither abstract nor virtual.

Inheritance is a central feature of object-oriented code, but it applies exclusively to reference types. Inheritance—as well as the features that support it, such as virtual methods—is not appropriate for value types, in part because of the way value type instances use memory.

Type Instance Lifetimes

Value types and reference types differ in the way each uses memory and, more specifically, in the lifetime of their instances. Value type instances are short-lived, and their lifetime is bound to the lifetime of the variables that represent them. For value types, the variable *is* the instance; when we create a new instance of a value type, the target variable effectively contains the instance data—that is, the value of each field of the type.

In many cases, the lifetime of a variable is defined by a block, such as a method body or a `foreach` loop. Any local variables within the block cease to exist when the block ends. Alternatively, a variable might be contained in another object, in which case the variable's lifetime is defined by the lifetime of the enclosing object. Whenever we copy a value type variable by assigning it to another variable or passing it as an argument to a method, the copy is a whole new instance of the type in a *different* variable.

Reference type instances, on the other hand, are generally long-lived and can be referred to by many variables. When we create a new instance of a reference type, we're given a reference to that instance in memory. Whenever we copy that reference, we're not also copying the instance. The original reference and the copy both refer to the same instance. References are stored in *reference variables*.

All reference type instances are allocated on the heap. Their lifetime is managed by automatic garbage collection, which releases their memory when they're no longer needed by the program. An object is considered unused when the garbage collector determines that no other live references to that instance exist. While reference type instances are not subject to their scope, reference variables *are* subject to scope, so when one goes out of scope, it's no longer a live reference to an instance. The lifetime of a reference type instance, then, is determined by the lifetimes of *all* the references to that instance.

A cost is associated with being allocated on the heap, because the garbage collection process takes time while the program is running. Ensuring that unused heap memory is properly cleaned up is a complex operation and may interrupt a program's normal execution for a short time, so an overhead is associated with reference types.

Value types don't require the overhead associated with garbage collection. The memory used by a value type instance can be freed when the lifetime of its variable ends. To understand lifetime a little better, let's look more closely at what we mean by *variable* in different contexts.

Variables

A *variable* is simply a named area of memory. We use this name—or *identifier*—to manipulate a memory location during the variable's lifetime. C# has five main kinds of variables:

Local variables

These are block-scope variables, where a *block* might be a method with a statement body, the body of a loop, or any section of code delimited by matching braces, {}. When control leaves a block at the closing brace, any variables that are local to the block go out of scope. When an exception is thrown in a block, the control flow also leaves that scope and any containing scope until the exception is caught or the program exits.

Instance fields

These are normal data members, known as *fields*, of structs and non-static classes. Each instance of a type has its own copies of any instance fields. The lifetime of an instance field is defined by the lifetime of the object to which it belongs.

Static fields

These fields are associated with a type, rather than individual instances of the type. The lifetime of a static field is normally tied to an application, so the instances associated with static fields are usually released when an application exits.

Array elements

Individual elements in an array are all variables. We can access a particular element by its index and alter the element instance if it is mutable.

Method parameters

The parameters in a method definition are technically called *formal parameters* but are commonly known as just *parameters*. A parameter's scope is the body of the method, exactly as if the parameter were declared as a local variable within the method's body. In code that calls a method, we pass *actual parameters*, better known as *arguments*, that correspond to the method's parameters.

Regardless of its kind, a variable always has an associated type. This might be an explicitly declared type, as in the declaration `int size`, or, for local variables, the type might be implied with the `var` keyword. If the variable's type is a reference type, the variable's *value* is a reference. A non-`null` reference is a handle to an instance somewhere on the heap. If the variable's type is a value type, the variable's value is an instance of the type.

Variables vs. Values

It's not always easy to intuit what counts as a variable and what counts as a value, but the distinction is important:

- *Variables* can be assigned to, although a `readonly` field variable can be assigned only within a constructor of the type of which it is a member, or using field initialization (which we'll discuss in "Field Initializers" on page 58).
- *Values* are the results of expressions—such as the result of calling `new`, the return value from a method, or a constant expression such as a literal number or `string` literal. Values *can't* be assigned to, but we use them to initialize variables by using assignment or passing them as arguments to method parameters.

Variables, for the most part, have names. Strictly speaking, individual array elements don't have their own names, but for an array variable `arr`, the expression `arr[`*`index`*`]` is essentially the element's identifier. A value can have a name but doesn't require one: the expression `2 + 2` produces a new value, but it is anonymous unless we assign that value to a variable.

The type of a value defines what an *instance* looks like. Among other things, the type might have multiple fields that need space allocated in memory when an instance of the type is created. The type of a variable defines the sort of value it can contain.

A value is just a pattern of bits. The type is a formal specification for interpreting that bit pattern to give it meaning in a program. Two values with identical bit patterns may be interpreted differently if they are different types. A pattern of bits that are all `0` means one thing if the type is `long`, but something else entirely if the type is `DateTime`.

A variable of value type directly contains its data, whereas a variable of reference type contains a reference to its data. More precisely, reference variables have a value that is a reference to an object somewhere on the heap. Put simply, a reference refers to an instance of a reference type; the value of a reference type variable is a reference.

The relationship between variables and values is that all variables *have* a value, although the value can't be accessed until the variable has been definitely assigned.

Definite Assignment

We can't read the value of a variable unless the compiler is satisfied that the variable has definitely been given an initial value. More formally, a variable can be read only after a value has been *definitely assigned* to it. The C# Language Specification precisely defines what constitutes definite assignment, but the essence is that a variable must have been assigned or initialized with a value at least once before its value is read.

If we try to obtain the value of any variable that hasn't been definitely assigned, the compiler raises an error to tell us that this isn't allowed. For example, when we declare a local variable within a method, it is uninitialized unless or until we assign a value to it. Such variables are initially considered *un*assigned. Conceptually, at least, an unassigned variable doesn't have a value.

When we assign something to a variable, we give that variable a new value. When we read from a variable, we obtain its value. Variables and values are both *expressions*, meaning we can evaluate them to produce a value, as long as they have been definitely assigned.

To reiterate, attempting to read a value from any variable that hasn't yet been definitely assigned is an error. When we use a `var` declaration for a local variable, we must provide an initial value where the variable is declared, because the type of the variable is inferred from the type of the value being assigned to it.

Instances and Storage

Now that we've clearly defined variables and values, we can explore how they relate to type instances. Whether an instance is a value type or a reference type affects where it is allocated and managed in memory; as a result, value type variables have some peculiarities that don't apply to references.

Value types do not always live on the stack, despite common misconceptions. Values for local variables are most often tied to the block scope of a method, and so might be associated with a stack frame for the method, but values can also be contained within another object as a member or an element in an array. Let's examine this more closely by looking at some examples of how variables are embedded in objects.

Embedded Values

If a variable is a field embedded within an instance of another type, its value is allocated within the memory for its enclosing object. This is especially important for value type variables that directly contain the instance of their type. Consider the `Color` struct in Listing 2-5.

```
public readonly struct Color
{
    public Color(int r, int g, int b)
        => (Red, Green, Blue) = (r, g, b);

    public int Red { get; }
    public int Green { get; }
    public int Blue { get; }
}
```

Listing 2-5: Defining a `Color` struct with multiple fields

The `Color` struct has three properties representing the components of an RGB color. When a `Color` value is used as a field or property in a class, an instance of that class will wholly contain a `Color` value on the heap. Take, for example, the `Brush` class in Listing 2-6, which has several fields, one of which is a `Color` type.

```
public class Brush
{
--snip--
    public enum BrushStyle { Solid, Gradient, Texture }

    private readonly int width;
    private readonly Color color;
    private readonly BrushStyle style;
}
```

Listing 2-6: A `Color` value embedded within the `Brush` class

The `Brush` type is a class and therefore a reference type. When we create an instance of any reference type, it's allocated on the heap. The `Brush` class has three fields, one of which is a `Color` instance, which itself has three fields (`Red`, `Green`, and `Blue`). An instance of `Brush` might look roughly like Figure 2-1 in memory.

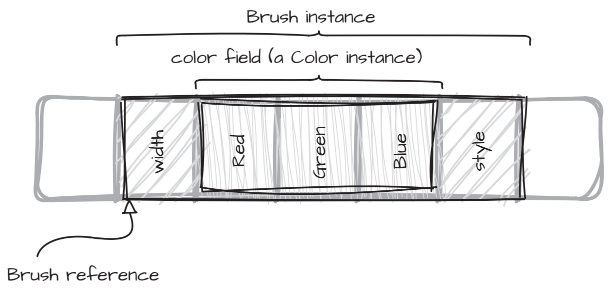

Figure 2-1: A `Color` value embedded in a `Brush` instance on the heap

When we create a new `Brush`, the instance is created on the heap and we're given a reference to it. The `color` field occupies memory directly within the memory space for the `Brush` instance. If we implemented `Color` as a record struct instead of a struct, the outcome would be the same. Record structs are value types in exactly the same way as structs and are allocated directly within the memory space of any enclosing object.

Value type instances are not individually garbage collected, but if a value type instance is embedded in another object that has been allocated on the heap, the *memory* used by the value type instance will be reclaimed during garbage collection of the enclosing object.

The lifetime of the `Color` instance represented by the `color` field is tied to the lifetime of the `Brush` instance. When the garbage collector determines that the `Brush` instance is no longer used, it will free up the memory for that instance, including the embedded `Color` value.

Array Elements

When a value type instance is an element in an array, it isn't (strictly speaking) a field of the array object, but the value is still embedded within the memory for the array. Arrays are always allocated on the heap, regardless of the type of their elements. When we create an array, we're given a reference to it. To illustrate, consider this array of `Color` values, where `Color` is a struct:

```
var colors = new Color[3];
```

The `colors` variable here is a reference to an array of three `Color` instances on the heap. The memory layout of the `colors` array might look like Figure 2-2.

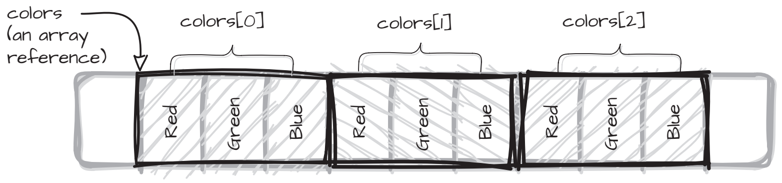

Figure 2-2: An array of `Color` structs in memory

In the `colors` array, each element is large enough to store the three `int` backing fields. If the element type had more fields, each element would require more space on the heap. If the `Color` type were a record struct rather than a struct, the layout would be identical; recall that the compiler translates record structs into structs.

Reference variables, by contrast, are all the same size, regardless of the number of fields declared in the type definition. The memory required for an array of references is determined only by the number of elements, not the size of each instance.

Whether the elements of an array are references or value type instances, the array is always on the heap, and the array variable refers to its elements. If the garbage collector determines that the array is no longer in use—that is, no live reference variables to it exist—then the memory for all of its elements is freed in one go.

Embedded References

Reference fields are also embedded in their enclosing type, but their instances are not. If we had implemented `Color` as a reference type in Listing 2-5, rather than a value type, the layout of a `Brush` instance would be somewhat different. The `color` field of the `Brush` class would be a reference, as illustrated in Figure 2-3.

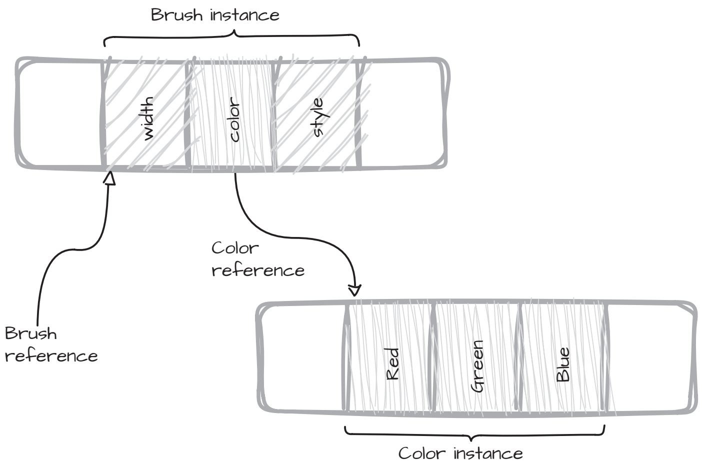

Figure 2-3: A `color` reference field embedded in a `Brush` instance

Instead of containing the entire instance of `Color` within its own memory, the `Brush` type's `color` field refers to a separate `Color` instance somewhere else on the heap. Reference type instances are always allocated on the heap and are independent of one another. This applies to any reference type, so it would be true if we implemented `Color` as either a class or a record.

The lifetime of the `Color` instance here is *independent* of the `Brush` instance. When the `Brush` instance is no longer used and its memory is released, the

`Color` instance will remain in memory until the garbage collector determines that it's no longer needed.

Field and Property Layout

All user-defined types can contain instance fields and properties. However, structs and record structs have one restriction that does not apply to classes or records: a value type definition can't embed a field of its own type.

You've already seen how value type instances directly contain their fields. If a type has a field that is itself a value type, that field also directly contains *its* data. If the type of that field is the same as its containing type, the compiler is unable to determine how to create it. Consider the simple struct in Listing 2-7 that embeds an instance of *itself* as a field.

```
struct Node
{
    Node p;
}
```

Listing 2-7: A struct containing an instance of itself

This example will not compile. The compiler can't know how to lay out the contained field named `p`, because `p`'s type isn't fully defined at the point where it is declared. The same is true of properties, because even automatic properties require a backing field, though that field is hidden from us.

The same reasoning applies to an indirect dependency, illustrated in Listing 2-8.

```
struct Tree
{
    Node root;
}
struct Node
{
    Tree leftChild, rightChild;
}
```

Listing 2-8: A struct with a cyclic dependency

Neither the `Tree` type nor the `Node` type can be created here because the layout of each depends on the other. This might sound draconian, but in practice it's rarely a problem, and we have an easy workaround: if we change the definition of either `Tree` or `Node` to make it a reference type, the compiler will accept this code. The rule applies only to value types because, as mentioned previously, references are always the same size regardless of the type to which they refer. This means the compiler doesn't need to know the layout of a class or record to establish a reference to it.

Boxed Values

References can refer only to objects on the heap and can't refer to individual value type instances, even those enclosed within a reference type object. The only way for a reference variable to individually refer to a value type instance is to make a copy of the value, put that copy on the heap, and refer to the copy with a new reference. The process of creating a copy and storing it on the heap, known as *boxing*, is automatic when the type of the variable is a reference type. A boxed value can always be converted back to its original value type, a process called *unboxing*, where the value contained in the box is *copied* into the target variable.

Boxing happens automatically when we refer to a value using a reference variable such as `object`, or when we pass a value as an argument to a method that takes a reference type parameter. Unboxing is always *explicit*: we need to cast the boxed variable back to its correct value type, as shown in Listing 2-9.

```
public readonly struct Color
{
    public Color(int r, int g, int b)
        => (Red, Green, Blue) = (r, g, b);

    public int Red { get; }
    public int Green { get; }
    public int Blue { get; }
}

var red = new Color(0xFF, 0, 0);
var green = new Color(0, 0xFF, 0);

❶ object copy = green;
Assert.That(object.Equals(❷ red, copy), Is.False);

var copyGreen = ❸ (Color)copy;
```

Listing 2-9: Boxing and unboxing

The type of the `copy` variable is `object`, and is therefore a reference, so the value of `green` gets boxed into `copy` ❶. Similarly, calling the `object.Equals` method boxes the value of `red`, because the method takes two `object` parameters ❷. We don't need to explicitly cast the value to the `object` type; it's boxed implicitly. We do require an explicit cast to unbox the value stored in `copy` into a new variable ❸.

As you'll see shortly when we cover the Common Type System, `object` is the base class of every type, meaning we can always use `object` to refer to any other variable, including value type instances. A struct can also implement one or more interfaces. Interfaces are reference types, so if we use either `object` or an interface type to refer to a value, that value is automatically boxed onto the heap.

A boxed value can be unboxed only to its original type. We can't, for instance, unbox an `int` value into a `double`, even though an implicit built-in conversion exists from `int` to `double`. If we attempt to unbox a value to anything other than its original type, we'll get an `InvalidCastException` at run time.

Boxed values are copied to the heap, which means the box is no longer subject to the scope of its variable and may exist beyond the lifespan of its original value. It's up to the garbage collector to clean up boxed values. Chapter 4 discusses boxing in more detail.

Semantics and Type

Value types have semantic implications that go beyond being an instance of a struct or record struct. Choosing a value type instead of a reference type when we define our own types requires much more than a consideration of possible optimizations. Records, in particular, differ from classes, because even though records are compiled into classes and are therefore reference types, they share some important behavioral characteristics with value types.

Before delving into the behavior of record and record struct types, we need to better understand how structs differ from classes.

The Common Type System

C# has a hierarchical type system, known as the *Common Type System*, in which all types derive from `object`, a keyword alias for the `System.Object` type. This is why we can always use `object` to refer to any other variable—although, as you just saw, in the case of value types, the instances are boxed so they can be referred to by `object` references.

Even the built-in types, such as `int` and `float`, inherit from `object`. In fact, all built-in types are aliases for types in the `System` namespace. The `System` types that underlie the numeric types are all structs and therefore value types. For example, `int` is an alias for the `System` type `public readonly struct Int32`.

Enumeration types created with the `enum` keyword are not aliases to `System` types, although they all *derive* from the `System.Enum` class. The individual values of an `enum` declaration have an underlying numeric type, which by default is `int`. We could specify a different numeric type—for example, if we wanted to allow the `enum` elements to have values larger or smaller than is permitted for an `int`.

The non-numeric built-ins `string` and `object` are aliases to classes in the `System` namespace, so they're both reference types.

When we use the `class` or `record` keyword to define our own reference type, our new type derives directly from the `object` base class unless it explicitly inherits from another type. The `object` base class is neither an interface nor `abstract`. It has a mix of virtual, nonvirtual, and static members, which provide the default implementations common to all objects.

All struct types (including record structs) and the `System.Enum` type implicitly derive from `System.ValueType` (for which there's no keyword alias), which in turn derives from the `object` base class, so all struct types derive *indirectly* from `object`. Value types, unlike reference types, have an intermediate base class defined by the language.

NOTE *`ValueType` itself is* not *a struct, which is sometimes overlooked. All structs implicitly inherit from `ValueType`, so `ValueType` itself must be a class. Moreover, `ValueType` is an* abstract *class, meaning we can create an instance of `object` but not of `ValueType`.*

The `ValueType` class overrides all the virtual methods defined in the `object` base class—`Equals`, `GetHashCode`, and `ToString`—and customizes their implementations to provide behavior tailored for value types. The `ValueType` implementations for `Equals` and `GetHashCode` are extremely important because they provide the *value-based* definition of equality that distinguishes value types from reference types. The difference between these implementations has to do with the way values are copied.

Copy Semantics

The difference between where reference types and value types store their instance data has important implications when we copy variables, because copying a reference does not copy the instance. Listing 2-10 shows a simple example to illustrate the difference.

```
  var thing = new Thing { Host = "Palmer" };
❶ var copy = thing;
❷ copy.Host = "Bennings";
  Assert.That(thing.Host, Is.EqualTo("Palmer"));
```

Listing 2-10: Copying a variable

Here we're copying the value of the `thing` variable into a new variable called `copy` ❶. Then we assign a new value to the `Host` property of `copy` ❷. The test checks that the properties of the original variable haven't changed. The success of the test assertion depends on whether `Thing` is a value type or a reference type.

As noted earlier, all variables have a value that we may copy to a new variable. If `Thing` is a value type, any copy we make is a new instance of the type, so if we modify any fields of that copy, those changes have no effect on the fields of the original value. Therefore, if `Thing` is a struct or a record struct, the test will pass.

If `Thing` is a reference type, on the other hand, the `thing` variable's value is a reference. When we copy a reference, only the value of the reference is copied, and it refers to the same instance as the original variable's value. This means if we modify the instance using one reference, that change is reflected in *all* the references to it. Thus, if `Thing` is a class or a record, the test will fail.

Locks and Reference Semantics

Some situations require the behavior of reference type variables, and using a value type instance would be incorrect or even disallowed. For example, we can't use a value type in a `lock` statement to prevent a section of code from being executed concurrently by multiple threads. The compiler forbids it because the variable used as a lock needs to be a reference to an object on the heap. The purpose of locking an object is to allow only a single thread to execute the code it protects at any given time. The object instance identifies the lock and can then have multiple references to it from different threads.

The underlying mechanism for the `lock` statement is the `System.Threading.Monitor` class. The `lock` statement translates to the `Enter` method of `Monitor`, which takes `object` as its parameter. Any instance of a value type passed to `Monitor.Enter` as an argument will automatically be boxed. Each thread calling `Monitor.Enter` will box the value separately, and the acquisition of the lock would never fail, rendering it pointless.

When we've finished with the lock, we need to call `Monitor.Exit` and pass the *same* reference used to acquire the lock with `Monitor.Enter`. The compiler inserts the code to call `Monitor.Exit` at the closing brace of a `lock` block. If we use a value type, the call to `Exit` will result in a new boxed value on the heap, and so will be a different reference to that used in the call to `Enter`. The result is that releasing the lock will fail with a `SynchronizationLockException` error.

This is one situation actively requiring reference semantics, because passing a reference to the `Enter` method doesn't copy the instance. The `monitor` and the code using the lock both have a reference to the same instance.

Identity Equality vs. Value Equality

When we say we're comparing variables to see whether they're equal, what we really mean is that we're comparing the variables' *values*. If two variables have the same value, they're considered equal. The type of each value plays an important role: the values being compared must be the same type, although one or both values may have resulted from an implicit conversion.

If we compare the values of two variables of the same reference type, their respective values are references, which compare equal by default if they both refer to the same object in memory. This is known as an *identity comparison*. We can override the default identity comparison behavior in our own reference types (a topic we'll examine in detail in Chapter 5), but two references to separate instances that have identical field values compare *unequal* according to the default identity comparison because they refer to different objects.

By contrast, two value type instances compare equal—again by default, because we can modify this behavior—if all the fields of one compare equal with their counterparts on the other. The difference in equality comparison behavior between value type instances and reference type instances is directly related to their respective copy semantics. Since a copy of a value type instance is a new independent instance with identical *state*, an identity comparison makes no sense. The two concepts of copying and equality are therefore intimately related.

The ability to compare two values to see whether they are equal is often underappreciated. Even if we rarely need to compare variables in our own code, commonly used classes such as `List<T>`, `Dictionary<T>`, and the LINQ methods that work on collections may be making those comparisons out of sight. `Equals` is a virtual method defined by the `object` base class, which is a clue to how fundamental it really is, because it means we can call the `Equals` method on *any* value to compare it with any other.

However, the `object.Equals` implementation always performs an identity comparison, which, again, is pointless for value types. For this reason, all structs implicitly inherit the `ValueType` class. `ValueType` overrides the `Equals` method to perform a value-based comparison.

The difference between what equality means for reference types and value types affects the way our code behaves at run time. Consider Listing 2-11, where the `Thing` type has not yet been allocated as a reference type or value type and does not explicitly override the `Equals` method. Here, we create two instances of `Thing` with the same value for their `Host` property. What happens when we call `Equals` depends entirely on whether `Thing` is a class, record, struct, or record struct.

```
public ??? Thing
{
    public string Host { get; set; }
}

var thing = new Thing { Host = "Palmer" };
var clone = new Thing { Host = "Palmer" };

Assert.That(clone.Equals(thing), Is.True);
```

Listing 2-11: Comparing equality of two independent variables

This assertion will fail if `Thing` is a class, because the `object.Equals` method will return `true` only if both `clone` and `thing` are references to the same instance, and they're not. The assertion will pass if `Thing` is a struct, because the `ValueType` implementation of `Equals` returns `true` if both `clone` and `thing` have the same value; that is, all their fields compare equal.

The `clone` and `thing` variables also compare equal if `Thing` is either a record or a record struct because they also use a value-based comparison for equality.

Records, Structs, and Value Semantics

Records are reference types but have value-like behavior when it comes to comparing two record variables for equality. When a record type is compiled, the compiler generates a class definition with an overridden implementation of the `Equals` method unless we define one ourselves. The `Equals` method generated for records compares two instances to determine if they have the same state, rather than just comparing two references to determine if they refer to the same instance.

In a struct, on the other hand, if we don't override `Equals`, the equality comparison relies on the implementation of `Equals` provided by the `ValueType` base class. Records, as reference types, don't inherit from `ValueType`. Record structs do inherit from `ValueType`, but, as with records, `Equals` is overridden by a compiler-generated implementation, because `ValueType.Equals` might not be the optimal implementation.

The `ValueType` implementation is necessarily general; it must work for *any* struct type, regardless of the types of the struct's fields. If a field of the type has a custom implementation of `Equals`, instances of the containing type must use that field's implementation for comparisons; a simple structural or bitwise comparison of the instances may not always be correct. The implementation of `Equals` provided by `ValueType` relies on reflection at run time to determine how to compare the fields and will use an overridden implementation of `Equals` to compare a field if the type of that field has one.

If we want to avoid the overhead of reflection in a struct, we must override `Equals` with our own implementation to compare each field and property with its corresponding field or property in the instance being compared. If each field and property value compares equal, using its `Equals` method where required, then the two instances are equal. This is essentially the implementation provided by the compiler for records and record structs.

To reiterate, structs, records, and record structs all employ a value-based comparison of their state to implement the `Equals` method, but for records and record structs, the implementation is generated automatically by the compiler, freeing us from the responsibility of providing our own custom implementation.

The variables we use for records—but not record structs—are references, and when we assign one record reference to another variable, we still get two references to the same record instance, just as we do if the type is a class. Records therefore have reference semantics for copying and value semantics for equality comparison.

The different comparison and copy semantics for value types and reference types have important consequences for the way instances of those types behave at run time. However, important differences also exist in the way those instances are created in the first place. In the next section, we'll look at how construction and initialization differ depending on whether the type of the instance is a value type or reference type.

Construction and Initialization

Creating a new object is superficially a simple operation, but behind the scenes the compiler goes to a great deal of trouble to make the process as efficient as possible. In principle, creating an object involves allocating the memory for an instance of a type and then calling a constructor whose job is to initialize the instance's fields. The syntax is identical for both value types and reference types, but `new` treats them differently and hides some complexity around how and where different types are allocated in memory.

In other words, the new expression is an abstraction that shields us from the implementation details of how memory is allocated and used.

Specifically, the memory for reference type instances is allocated dynamically. When a new instance of a class or record type is created, the memory is allocated on the heap at run time. Instances of struct and record struct types are allocated differently, depending on how the resulting instance is used. Consider this code, which initializes a variable with a new instance of a type named Thing:

```
var thing = new Thing();
```

This basic syntax for creating an object and assigning it to a variable is the same whether Thing is a class, struct, record, or record struct. As you'll see over the coming sections, this code depends on Thing having an accessible constructor that can be invoked with no arguments, which isn't necessarily the case when Thing is a reference type. For the time being, though, let's assume that Thing instances can be created this way. If Thing is a class or a record, new causes memory to be allocated on the heap at run time and returns a reference to the new object, which is assigned to the thing variable.

If Thing is a struct or a record struct, the new instance is assigned to the thing variable. However, this code may or may not allocate memory for a new instance of Thing and may or may not call a constructor. The reason is that construction and initialization are separate processes. Part of the difference is related to whether a Thing is a value type or reference type.

Default Initialization

Default initialization means that each of a type's fields, including the backing fields for properties, is given a default value, which is defined in the language to mean one of the following:

- References are set to null.
- Built-in numeric value type variables are set to 0.
- All other value types are default-initialized.

Default-initialized reference type fields are a common cause of errors. For example, the simple MusicTrack struct in Listing 2-12 relies on us manually initializing an instance by setting its properties. If we neglect to set suitable values for the properties of a MusicTrack instance, we may be rewarded with an exception when we use the instance.

```
public struct MusicTrack
{
    public string Artist { get; set; }
    public string Name { get; set; }

    public override string ToString()
        => $"{Artist.ToUpper()}: {Name.ToUpper()}";
}
```

```
var defaultTrack = new MusicTrack();

var print = defaultTrack.ToString();
```

Listing 2-12: Initializing reference type fields

The call to `ToString` causes a `null` reference exception because the `defaultTrack` value has been default-initialized. The `ToString` method calls `ToUpper` on its `Artist` and `Name` properties, whose default-initialized value is `null`. We need to be alert to any uses of default-initialized references in order to avoid such problems resulting from accessing a `null` reference. One way to minimize the impact of default-initialized values is by providing our own instance constructors.

Instance Constructors

An instance constructor, like a method, can have zero or more parameters. Also like methods, constructors can be overloaded, so we can define several constructors for a type, each with a different number of parameters, or parameters of different types. Constructor definitions for classes, structs, records, and record structs have many similarities, but several important differences exist.

In Listing 2-13, we add a constructor for the `MusicTrack` struct and use the parameter values to initialize the instance's property values. We use the null-coalescing operator `??` to assign an empty string for each property if its corresponding parameter is `null`.

```
public readonly struct MusicTrack
{
    public MusicTrack(string artist, string name)
    => (Artist, Name) = (artist ?? string.Empty, name ?? string.Empty);

    public string Artist { get; }
    public string Name { get; }

    public override string ToString()
        => $"{Artist.ToUpper()}: {Name.ToUpper()}";
}
```

Listing 2-13: Adding an instance constructor with parameters

By adding a constructor, we no longer have to rely on `MusicTrack` users setting the properties explicitly, since the initial values for those properties are set in the constructor. We have made those properties get-only—that is, they can be given a value only in the constructor—and made `MusicTrack` a `readonly` struct. However, we must still be cautious of using the property values inside the `ToString` method because instances of any value type can always be default-initialized, regardless of the presence of a user-defined constructor definition. Adding our own constructor for `MusicTrack` to give meaningful values to the properties isn't sufficient protection against exceptions that occur from calling methods using a `null` reference, because `MusicTrack` is a struct type.

If the nullable reference type feature is enabled (see "Nullable Reference Types" on page 64 for more), the constructor's parameters will be non-nullable variables, meaning that passing `null` for either argument would cause a compiler warning. Using non-nullable parameters doesn't mean that `null` can't be passed as an argument, but we may decide that the warning is sufficient protection, potentially allowing us to omit the null-coalescing assignments in the constructor. The nullable reference type feature doesn't, however, mean we can avoid verifying that the property values are not `null` prior to using them in the `ToString` method. Fortunately, the null-conditional operator makes the check straightforward and safe:

```
public override string ToString()
    => $"{Artist?.ToUpper()}: {Name?.ToUpper()}";
```

Here the presence of the null-conditional operator, a `?` appended to each property name, means that in each case the `ToUpper` method will be called only if the property is a non-`null` value. If either property is `null`, the result of the expression between the braces within the string is `null`, which the string interpolation treats as an empty string.

If `MusicTrack` were a class or record, the presence of our own constructor would mean we could no longer create an instance without passing arguments like this:

```
var track = new MusicTrack();
```

If we attempt to create a default-constructed instance, we get the following compiler error:

```
[CS7036] There is no argument given that corresponds to the required formal parameter 'artist'
of 'MusicTrack.MusicTrack(string, string)'
```

If we don't provide any constructors for a class or record, the compiler inserts a default constructor for us. If we define a constructor when we define our own reference type, however, the compiler will not generate the default constructor. The compiler doesn't create a default constructor for value types, but an instance of a struct or record struct can be default-initialized whether or not we define our own constructor.

Default and Generated Constructors

The behavior of reference types and value types differs partly because reference types are allocated on the heap, but value types might not be. The compiler generates a default constructor for reference types because instances of such types are allocated dynamically, and their instances are initialized at run time. When a reference type instance is allocated on the heap, the memory for it is set to zero, effectively default-initializing the instance.

Value types are treated differently because their memory isn't necessarily allocated at run time: for local value type variables, the compiler may *reserve* memory for the instance data, and the program accesses that memory directly. The underlying Common Intermediate Language (CIL) has an efficient instruction for default-initializing value type instances that effectively zeroes out the memory used by the instance, wherever its memory actually resides.

We can think of the default initialization of a struct or record struct as being performed by a compiler-provided default constructor, because the result is identical in any case. Default-initializing value types offers a minor performance advantage because it doesn't require a method call to a constructor, although it's almost never the most significant optimization.

In a positional record or a positional record struct, the compiler generates a public constructor based on the parameters we use in the type definition, like this:

```
public sealed record Color(int Red, int Green, int Blue);
```

The parameters to `Color` in this example tell the compiler to create public properties using those names and their types. The compiler also creates a constructor with the same signature as the record's parameter list, where the properties are assigned their values. The constructor generated by the compiler is the equivalent of this:

```
public Color(int Red, int Green, int Blue)
    => (this.Red, this.Green, this.Blue) = (Red, Green, Blue);
```

Although the constructor has been generated by the compiler, it's still considered a user-defined constructor and therefore still suppresses the default constructor for the `Color` record.

Regardless of its type, an instance is always default-initialized when it's first created, whether its memory is being allocated on the heap or elsewhere.

When we define our own constructor for a class, we can rely on all the fields having been default-initialized prior to the constructor's body; the fields of a class are considered *initially assigned* within the constructor. In a struct's constructor, the fields are *initially unassigned*, so we must definitely assign a value for every field of a struct or record struct, even if it's simply to replace the value with its default-initialized equivalent.

Overloaded Constructors

We can provide a constructor with parameters for any type, and we can overload the constructor by defining several constructors that have different numbers or types of parameters. This is useful when we want to support different ways to construct our type. For instance, Listing 2-14 shows a struct that has two constructors with differing signatures.

```
public readonly struct Color
{
    public Color(int red, int green, int blue)
        => (Red, Green, Blue) = (red, green, blue);

    public Color(uint rgb)
        => (Red, Green, Blue) = Unpack(rgb);

    public int Red { get; }
    public int Green { get; }
    public int Blue { get; }
}
```

Listing 2-14: Overloading constructors

The first constructor initializes the three properties from three separate parameters (`red`, `green`, `blue`). The second constructor receives a numeric representation of an RGB value and initializes the `Red`, `Green`, and `Blue` properties by calling the `Unpack` method (not shown here) to unpack the number into its component parts. We select the different overloads when using the constructor by passing different arguments, as shown in Listing 2-15.

```
var orange = new Color(0xFFA500);
var yellow = new Color(0xFF, 0xFF, 0);
```

Listing 2-15: Selecting the correct overload

Here, the `orange` variable is created using the constructor with a single `uint` parameter (the second constructor in Listing 2-14), and the `yellow` variable uses the constructor with three `int` parameters (the first constructor in Listing 2-14).

Parameterless Constructors

As noted earlier, defining our own constructor for a class type will inhibit the compiler-generated default constructor, meaning that we can create new instances of the type only by passing arguments to our own constructor's parameters. If we need to create instances of such a reference type without arguments, we can define our own *parameterless constructor*, which we might use to initialize reference type fields and properties to non-`null` values. This is common when a class contains a collection that needs to be initialized but can be empty, as demonstrated in Listing 2-16.

```
public sealed class Playlist
{
    public Playlist(IEnumerable<MusicTrack> items)
      ❶ => queue = new(items);

    public Playlist()
      ❷ => queue = new();
    public void Append(MusicTrack item)
        => queue.Add(item);
--snip--

    private Queue<MusicTrack> queue;
}
```

Listing 2-16: Defining a parameterless constructor

The two constructors defined here allow us to create a `Playlist` either by passing a sequence of items to populate the queue ❶ or by passing no arguments ❷. If we pass no arguments, the `queue` field is initialized as an empty queue, ensuring that it isn't `null`.

Both constructors initialize the `queue` field by using type inference, a feature called *target-typed new*, introduced in C# v9.0. The compiler deduces the type required by `new` from the type of the target variable being initialized—in this example, a `Queue<MusicTrack>`. The `queue` field is guaranteed to be non-`null` for any `Playlist` instance, so we don't need to check for `null` in the `Playlist.Append` method.

In a positional record, the compiler creates a constructor based on the positional arguments for the record, so by default, instances of a positional record can't be created without arguments. We can define our own parameterless constructor for a positional record if we require that behavior. A struct or positional record struct, on the other hand, can *always* be created without arguments, whether or not we define our own constructors.

Structs and Default Values

As of C# v10.0, we can define our own parameterless constructors for value types to help ensure that any reference fields are non-`null`. However, we still need to check for `null` in a value type's implementation because an instance of a struct or record struct can always be default-initialized, effectively bypassing any constructors we define. This is illustrated in Listing 2-17, where we add a parameterless constructor for the `MusicTrack` struct that explicitly initializes the two `string` properties.

```
public readonly struct MusicTrack
{
    public MusicTrack()
        => (Artist, Name) = (string.Empty, string.Empty);

    public MusicTrack(string artist, string name)
        => (Artist, Name) = (artist, name);

    public string Artist { get; }
    public string Name { get; }

    public override string ToString()
        => $"{Artist?.ToUpper()}: {Name?.ToUpper()}";
}
```

Listing 2-17: Adding a parameterless constructor for a struct

The parameterless constructor sets both reference type properties to a non-null value, so calling ToUpper on either property is safe when we're using a MusicTrack instance that was created using new MusicTrack. However, this doesn't mean we can omit the null-conditional checks in ToString. It's still possible for Artist or Name to be null if the instance is a default-initialized MusicTrack—for example, when it's an element in an array:

```
var favorites = new MusicTrack[3];

var print = favorites[0].ToString();
```

Without the checks for null in ToString, this code would cause ToString to throw a NullReferenceException because the creation of the favorites array doesn't call our parameterless constructor on its elements. Each element is default-initialized, leaving the Name and Artist properties with their default value of null, so attempting to call the ToUpper method on a null reference causes the exception.

Array elements are default-initialized without invoking any parameterless constructor we provide. The parameterless constructor is reserved for when we create a new instance by using the new keyword.

Value Type Initialization

One quite subtle consequence of the way value type instances are allocated in memory is that if a value type's fields are all public, we can definitely assign a value for each field outside the constructor (as long as they're not read-only), which results in the whole instance being fully assigned.

For example, Listing 2-18 assigns a value to each field of an uninitialized struct variable.

```
public struct Color
{
    public int red;
    public int green;
    public int blue;
}

Color background;   // initially unassigned variable

background.red = 0xFF;
background.green = 0xA5;
background.blue = 0;

Assert.That(background.red, Is.EqualTo(0xFF));
```

Listing 2-18: Definitely assigning a struct

This code compiles, and the test passes. We can read the value of the red field, even though we've never allocated the background variable with new or invoked a constructor for it. The same would be true if Color were a record struct instead.

This example demonstrates that value type variables directly contain an instance of their type. Assigning to each field means we don't need to explicitly construct an instance. However, relying on this behavior is likely to cause other problems, not the least of which is that using public fields leaves the `Color` type open to misuse, intended or not. In practice, a constructor is a much better way to initialize a value type's fields, which should all be private and read-only.

Note that if we alter the public fields to be publicly mutable properties, this code will fail to compile. We can't access a property of a value type in any way until the instance itself has been fully, and definitely, assigned. Every property has a backing field generated by the compiler, and that backing field is always private.

Constructor Accessibility

Constructors with parameters can be made public or private in any type. Private constructors are useful when we want to prevent users from creating instances with certain arguments. We used this technique in "Static Creation Methods" in Chapter 1 to force users to call the static class factory methods we defined in order to create certain values, rather than using the `new` keyword directly. In a class or record, we can make the parameterless constructor private to prevent users from creating default-constructed instances, shown for the `Color` record in Listing 2-19.

```
public sealed record Color
{
    private Color() { }

    public static Color Black { get; } = new Color();


--snip--
}
```

Listing 2-19: Making constructors private for reference types

Since the constructor for `Color` is marked `private`, we can use it to initialize the static `Black` property value and any other static or instance members of `Color`, but it's inaccessible to code outside of the `Color` type. If users of `Color` forget and attempt to create an instance with `new`, the compiler forbids it:

```
var black = new Color();

[CS0122] 'Color.Color()' is inaccessible due to its protection level
```

Classes and records can also use the `protected` keyword on a constructor, making it available to inheriting types. Since structs and record structs can't be inherited, the compiler will prevent the use of `protected` in a value type.

In a struct or record struct, if we define our own parameterless constructor, it *must* be public. Struct and record struct instances can *always* be default-initialized, whether or not we provide a parameterless constructor.

Field Initializers

In a class or record definition, and in structs or record structs after C# v10.0, we can assign initial values to fields inline by using *field initializers*. We can do the same with automatic properties by using *property initializers*, which initialize the hidden backing field associated with the property. Listing 2-20 uses a field initializer for the `queue` field of the `Playlist` class from Listing 2-16 to assign an initial value and adds a `Name` property for `Playlist` that we also assign an initial value by using a property initializer.

```
public sealed class Playlist
{
--snip--

    public string Name { get; set; } = "_playlist";

    private Queue<MusicTrack> queue = new();
}
```

Listing 2-20: Assigning initial values for fields and properties

Field and property initializers are part of object construction but are not applied when a value type instance is being default-initialized. Conceptually, initializers are applied just before the body of a constructor. As noted previously, the compiler creates a default constructor for class and record types if no user-defined or positional constructors are present; however, the compiler won't synthesize a parameterless constructor for any value type. Therefore, if we want to use field or property initializers for struct or record struct types, we must also define at least one constructor of our own. This can be a parameterless constructor or a constructor taking one or more parameters.

Field initializers can't reference any instance members. However, since static fields are guaranteed to be definitely assigned before any instance fields, a field initializer can reference a static value. Static fields can also have initializers and can reference other static fields. However, we need to take care when referencing one static field from another static field because they're initialized in the order in which they appear in the class.

Object Initializers

With *object initializers*, we set values for publicly mutable properties of a variable at the point of creating a new instance, like this:

```
var fineBrush = new Brush { Width = 2 };
```

Classes, records, structs, and record structs accept this syntax, and they all behave the same way. The initialization process is the same for each: a constructor is invoked in the usual way to create an instance, and then the value is assigned to the property of the instance. In this example, a `Brush` is created using a parameterless constructor (or one with all-optional parameters), but we can call any constructor before the initialization expression inside the braces. In the special case of a constructor that requires no arguments, we can leave out the parentheses for the constructor.

Classes and records require an accessible parameterless constructor to use this syntax. If the parameterless constructor of a class or record is hidden or nonpublic, we *must* invoke a valid constructor before the object initialization within the braces. We don't have to worry about this for struct or record struct types because they can always be default-initialized if the type has no parameterless constructor.

init-Only Properties

As of C# v9.0, any property can be *init-only*, meaning it can be written to only during the creation of a new instance. Prior to C# v9.0, object initialization required properties to have a public `set` accessor, meaning object initialization couldn't be used with immutable properties. Object initialization requires the value of the property to be set after the constructor has completed, which wasn't permitted for properties without a public `set` accessor. An `init` accessor allows a property to be set during object initialization and then makes the property immutable after the initialization is complete.

The `Color` struct in Listing 2-21 demonstrates how `init`-only properties are used during object initialization.

```
public readonly struct Color
{
    public int Red { get; init; }
    public int Green { get; init; }
    public int Blue { get; init; }
}
var orange = new Color { Red = 0xFF, Green = 0xA5 };

Assert.That(orange.Red, Is.EqualTo(0xFF));
Assert.That(orange.Green, Is.EqualTo(0xA5));
Assert.That(orange.Blue, Is.EqualTo(0));
```

Listing 2-21: Setting properties as `init`-only

When we create the `orange` variable, a new `Color` is first default-constructed, giving each property its default value of `0`. The object initializer between the braces gives new values to the `Red` and `Green` properties, leaving the `Blue` property with its default value. Note that `Color` is a `readonly` struct, which requires that the struct has no mutable properties.

We can assign a value to an `init`-only property in an instance constructor or by using object initialization, but we can't assign a new value after the instance has been created. An `init`-only property is immutable. The `init` accessor syntax can be used for properties and indexers for any type, although it was introduced in C# v9.0 to support a special initialization syntax supported by records and known as *non-destructive mutation*.

Non-destructive Mutation

Records and record structs support the non-destructive mutation syntax, and as of C# v10.0, so do structs and anonymous types. Syntactically, non-destructive mutation is similar to object initialization, except that it initializes a new instance by copying an existing one and providing new values for selected properties in that copy. Listing 2-22 demonstrates this syntax, using the `with` keyword to copy the `orange` record variable to a new variable named `yellow`, and then assigning a new value to one of the properties of the copy.

```
public sealed record Color(int Red, int Green, int Blue);
var orange = new Color(0xFF, 0xA5, 0);

var yellow = orange with { Green = 0xFF };

Assert.That(yellow.Red, Is.EqualTo(0xFF));
Assert.That(yellow.Green, Is.EqualTo(0xFF));

Assert.That(orange.Green, Is.EqualTo(0xA5)); // unchanged in orange
Assert.That(orange.Blue, Is.EqualTo(0));
```

Listing 2-22: Initializing a copy of a record with non-destructive mutation

The `with` expression we use when we create the `yellow` variable creates a new instance of the `Color` record with property values identical to the original `orange` instance. Those properties specified between the braces following `with` are then assigned the values by using the same syntax as object initialization. This approach is called *non-destructive* mutation because no changes are made to the original record.

Constructors and initializers are both ways we can create new instances with known values. However, sometimes we can't provide an initial value for a variable, but leaving it uninitialized is too restrictive: we can't even test it to see whether it has a value, owing to the rules governing definite assignment. In the next section, we'll examine the options open to us when we need a variable with no value, and how value types and reference types differ here too.

null Values and Default Values

A plain value type variable can never be `null`. An instance of a value type directly contains all of its fields, and there's not necessarily a representation of "no value." A default-initialized instance of a value type is not the same thing—it's a complete instance of the type, just with the default-initialized values for each of its fields.

We can employ a nullable value type, which can be assigned and compared with the value `null`, as you'll see shortly, but plain value type instances are incompatible with `null`. The `null` constant expression is a reference and therefore can be assigned only to reference variables. One of the implications of not being able to assign null to a value type variable is that we can't pass null as an argument to a value type method parameter.

Similarly, attempting to *compare* a value with null makes no sense. If we do, as shown in Listing 2-23, the compiler rejects the code.

```
public readonly struct Speed
{
--snip--
}

var c = new Speed();

Assert.That(c == null, Is.False);
```

Listing 2-23: Comparing a value type variable with `null`

The error from the compiler is shown here:

```
[CS0019] Operator '==' cannot be applied to operands of type 'Speed' and '<null>'
```

We can, however, compare any reference type with null, and, as of C# v8.0, we can use a constant pattern to make this comparison more direct by using the is keyword:

```
Assert.That(someObject is null, Is.True);
```

Comparing any value type with null makes no sense, whatever method we choose, because null is a reference and as such is represented differently than a value type. That said, the rule against comparing value types with null has one exception: generic types.

Generics and null

In a generic class or method, an unconstrained type parameter variable can be compared with null. An unconstrained generic type can be either a value type or a reference type. To illustrate, the simple example in Listing 2-24 compares an instance of a generic parameter type with null.

```
public static int Compare<T>(T left, T right)
{
    if(left is null) return right is null ? 0 : -1;
--snip--
}
```

Listing 2-24: Comparing a generic type parameter instance with `null`

The Compare generic method has a type parameter named T that might represent either a value type or a reference type, because it has no type constraints. In this instance, T is not known to be a value type, so the compiler allows the syntax. If T's type is determined at run time to be a value type, the whole expression simply evaluates as false.

The compiler still prevents us from assigning null to a variable of type T, because if T were a value type, the assignment would fail at run time. Similarly, we can't return null through an unconstrained type parameter, demonstrated in Listing 2-25.

```
public static T Consume<T>(IProducerConsumerCollection<T> collection)
    => collection.TryTake(out var item) ? item : null;
```

Listing 2-25: Trying to return null as a generic parameter type

This gives the following error:

```
[CS0403] Cannot convert null to type parameter 'T' because it could be a non-nullable value
type. Consider using 'default(T)' instead.
```

In this example, the difficulty arises because T is unconstrained. It might represent a struct or record struct type, for which null is not a valid value. The error message gives us a clue that instead of returning null, we can return a *default* value for T. Default values have other, more significant use cases too, but also some limitations.

Generics and Default Values

The concept of a default value is closely related to a null value, especially in the context of generic types and methods. At times, we—and the compiler—must ensure that an instance of a generic parameter type T is definitely assigned, even when T's type is not known at compile time. We can't just use new to make a new instance of type T because the compiler isn't able to determine which constructors are available for T.

If T is a value type, we can *always* make a default instance by using default initialization or by calling a parameterless constructor, but if T is a reference type, it might not have an accessible default or parameterless constructor. We can use the new constraint on T, meaning that our generic type or method will work only with types that have an accessible parameterless constructor, but this might be too restrictive.

In a generic type, we can use the generic parameter to denote a field or property of the generic parameter type. Generic value types must ensure that *all* their fields are definitely assigned before control leaves the constructor. To make that possible, we use the default keyword to initialize a default instance of T, as in the generic struct shown in Listing 2-26.

```
public readonly struct Node<T>
{
    public Node(int index)
    {
        idx = index;
        contained = default;
    }
    private readonly int idx;
    private readonly T contained;
}
```

Listing 2-26: Initializing a default instance of a type parameter

In the Node constructor, the contained field is assigned the default value of its type by using the target-typed default literal (available since C# v7.1), which is equivalent to the expression default(T). Where T is a class or record, its default value is null, and where T is a struct or record struct, the default value is a default-initialized instance. Note that initializing a value by using default does not invoke a parameterless constructor, if we have defined one. This code is valid because we can always create a default *value* for a variable of type T: if T is a value type, the value is a default instance of T, and if T is a reference type, a default T is null. The default keyword has many uses outside of generic types and methods, but within generic code it's indispensable.

Default values are useful, but they're not sufficient to identify a particular value type instance as invalid. In other words, we can't use a default when what we really mean is *no value present*. The default value of a struct or record struct is a default-initialized instance and might therefore be a *valid* value. Consider Listing 2-27.

```
int x = default;
int y = 0;

Assert.That(x.Equals(y), Is.True);
```

Listing 2-27: Default values can be valid.

The default value for an int type is 0, which we may use to indicate an invalid number in some circumstances but not all. Whether that matters, especially for our own value types, depends on the context in which instances of the type are used, but limiting valid integers to only nonzero values would be very restrictive. Fortunately, we have an alternative.

Nullable Value Types

Nullable value types allow us to have a representation of a value type that means *no value present*. A nullable value type is a wrapper around a value type, and a nullable value type variable may or may not have a value. A nullable value type variable can also be assigned the value null, demonstrated by using a simple test in Listing 2-28.

```
int? x = null;
int y = 0;

Assert.That(x.Equals(y), Is.False);
```

Listing 2-28: Using nullable values

The ? following the int type of the x variable is shorthand for saying that x is a Nullable<int>. We can now represent an invalid value for x that's distinct from any valid values for int. We can use a nullable variable for any value type, not just built-ins. The default value for a nullable is null, as shown here:

```
int? x = default;
int? y = null;

Assert.That(x.Equals(y), Is.True);
```

This test passes because x and y are both null. The declaration of x in the first line doesn't initialize a default int but rather a default Nullable<int>. Equality comparison between nullable values compares the underlying value if there is one. Two nullable values are equal if they both have no value, or values that themselves compare equal. Nullable<T> is a struct and overrides the Equals method to provide this behavior.

As a consequence of not being able to assign null to a plain value type variable, we can't use a plain value type on the right-hand side of an as expression, like this:

```
object speed = new Speed();
var actual = speed as Speed;
```

If Speed is a struct or record struct, this code won't compile, because if the cast fails, the as operator will return null. As we know, null can't be assigned to a value. The solution is to use a nullable value type as the source of the conversion, as shown here:

```
var actual = speed as Speed?;
```

The type of the actual variable is a nullable Speed in this example and will have the value null if the conversion fails—that is, if the speed variable is not in fact a Speed type.

Nullable Reference Types

C# v8.0 introduced *nullable reference types*, a feature that allows the compiler to warn us when a reference is or might be null and we expect it to have a real value. While reference variables have always been able to have a null value, the nullable reference type feature allows us to express whether we *intend* for them to. In other words, when we use a nullable reference type variable, we're being explicit about our intention that null is an *expected* potential value for a variable.

Reference variables are non-nullable by default. In the declaration in Listing 2-29, the brush variable is a *non-nullable* reference.

```
object brush = null;
```

Listing 2-29: Declaring a non-nullable reference variable

The compiler performs static analysis that enables it to issue a warning if a non-nullable reference can't be guaranteed to be non-`null`. To state that with fewer negatives, the compiler issues a warning if a value that may be `null` is assigned to a non-nullable reference. In particular, assigning `null` to a non-nullable reference, as we just did, provokes this warning:

```
[CS8600] Converting null literal or possible null value to non-nullable type
```

If we attempt to pass a possibly `null` value as an argument to a non-nullable method parameter, we'll get a warning from the compiler. Consider the method in Listing 2-30, which capitalizes the first character of each word in a string.

```
public static string ToTitleCase(string original)
{
    var txtInfo = Thread.CurrentThread.CurrentCulture.TextInfo;
    return txtInfo.ToTitleCase(original.Trim());
}
```

Listing 2-30: Defining the `ToTitleCase` *method with a non-nullable reference parameter*

Within the `ToTitleCase` method, we should be able to depend on the `original` parameter having a real, non-`null` value, because it's a non-nullable string. That means we can avoid explicitly writing code to check that it isn't `null`. When we call `ToTitleCase`, if the compiler can't guarantee that the argument we pass isn't `null`, it will give us a warning.

We might have a legitimate need for a `null` reference, however, in which case we mark the type of a variable as nullable to suppress the compiler warnings about possible null assignment. The syntax is the same as for nullable value types: we append a `?` to the type. Listing 2-31 shows a collection of nullable `string` elements designated by the `string?` type name.

```
var names = new List<string?>();
// Load names from somewhere, may contain null elements
--snip--
var properNames = names.Select(name => ToTitleCase(name));
```

Listing 2-31: Passing a possibly `null` *argument for a non-nullable parameter*

If we apply the `ToTitleCase` method from Listing 2-30 to this collection, we get a similar compiler warning as with Listing 2-29, where we explicitly assigned `null` to a non-nullable reference type variable:

```
[CS8604] Possible null reference argument for parameter 'original' in 'string
ToTitleCase(string original)'.
```

We're given this warning because the compiler can't guarantee that the collection contains no `null` elements. The compiler assumes any of the elements may be `null` because the element type of the collection is a nullable reference.

If we explicitly check each element before making the call to `ToTitleCase`, the compiler can determine that we're not using a `null` reference as an argument to the method. To achieve that, we could unpack the `Select` expression into a loop, such as the `foreach` loop in Listing 2-32.

```
foreach (var name in names)
{
    if(name is not null)
        properNames.Add(ToTitleCase(name));
}
```

Listing 2-32: Explicitly using a non-`null` reference

This code doesn't prompt a warning about the argument in the call to `ToTitleCase` because the compiler can perform enough analysis on the code preceding the method call to guarantee that the `name` argument isn't `null`.

However, sometimes the compiler needs our help to determine whether it's safe to assign a variable to a non-nullable reference or to call a method with a non-nullable parameter. Listing 2-33 shows a slightly modified version of Listing 2-31 calling `ToTitleCase`, where any `null` elements are filtered out before the method is called.

```
var properNames = names
    .Where(name => name is not null)
    .Select(name => ToTitleCase(name));
```

Listing 2-33: Removing `null` elements before the method call

This code gives us the same warning as in Listing 2-31, however, because the compiler can't be certain `ToTitleCase` won't be invoked with a `null` argument. Although it looks as if the check for `null` is being made inline, in fact we're calling a lambda function to make that comparison, and the compiler doesn't attempt to analyze every possible code path to make this safe. Fortunately, we have a workaround.

The Null-Forgiving Operator

We can use the *null-forgiving operator* to inform the compiler that we definitely know what we're doing and that no `null` references are used as arguments to a non-nullable parameter. The null-forgiving operator is an `!` appended to the variable, which is why it's also referred to as the *dammit operator,* as in, "It's definitely not `null`, dammit!" When we've filtered out all the `null` elements from our collection, we apply the dammit operator to the argument for `ToTitleCase`, as shown in Listing 2-34.

```
var properNames = names
    .Where(name => name is not null)
    .Select(name => ToTitleCase(name!));
```

Listing 2-34: Using the null-forgiving operator

Using the null-forgiving operator with the argument to `ToTitleCase` convinces the compiler that it is safe to call the method having a non-`null` reference type parameter. If we were to inadvertently pass a `null` reference, we'd (justifiably) get the dreaded `Object reference not set to an instance of an object` exception. We must take care when using the null-forgiving operator that we really do know that the variable can't be `null`.

Nullable reference types, while having the same syntax as nullable value types, are just a device that indicates to the compiler that we're making certain assumptions about the variable. Unlike nullable value types, which are underpinned by a distinct type with behavior injected by the compiler, nullable reference types are a purely compile-time mechanism, used for static analysis, and do not change the behavior of our code in any way. At run time, nullable and non-nullable references are just references. Nevertheless, distinguishing between them in code is useful for encoding our assumptions about nullability.

Unexpected `null` reference exceptions are the curse of many programs and a class of error that programmers everywhere go to great lengths to try to avoid. The nullable reference type feature of modern C# is one that shifts some of that responsibility away from the programmer and onto the compiler.

Summary

My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

—Tony (C.A.R.) Hoare

The type system in C# is broadly similar to many other programming languages, including its support for user-defined types. C# differs in its distinction between reference types and value types. Although there are various recommendations on when to choose to define a value type instead of a reference type, including documentation from Microsoft, those guidelines often take only part of the story into consideration.

The technical purpose of distinguishing value types from reference types is to allow the compiler and Common Language Runtime to make assumptions about values that may allow certain opportunities for optimization. Some of the differences we've discussed result from the way reference and value type instances are stored and managed in memory. That value type variables are not independently subject to garbage collection can itself be a big win. However, we can't just turn our classes into structs or record structs and expect that our programs will suddenly use less memory or run more quickly. Value semantics involves much more than just declaring something as a value type.

Likewise, the copy-by-value behavior of value types is more than just a side effect of the way values use memory. Copying by value gives rise to many of the constraints that are imposed on value types and for which reference types have no need. Using value types where they're appropriate can make our code clearer and simpler in subtle ways, like not having to check for `null` values on every use of a value. The characteristics of copying values also affect the behavior of the `Equals` method; although comparing variables to see if they are equal may sound inconsequential, it's an essential aspect of working with variables.

The distinction between value types and reference types, then, is not just a list of *restrictions*. Genuine semantic differences affect our programs' behavior and can bring tangible benefits. One advantage of value types is that they can never be `null`. Constantly having to check references to ensure that they're valid can be tiresome and error-prone. Using the non-nullable reference type feature is one way we reduce the occurrence of unexpected errors arising from dereferencing a `null` reference.

One of the great strengths of C# being a compiled and type-safe language is that the compiler can identify many kinds of errors *before* our program is ever run.