.NET Learning

# Types

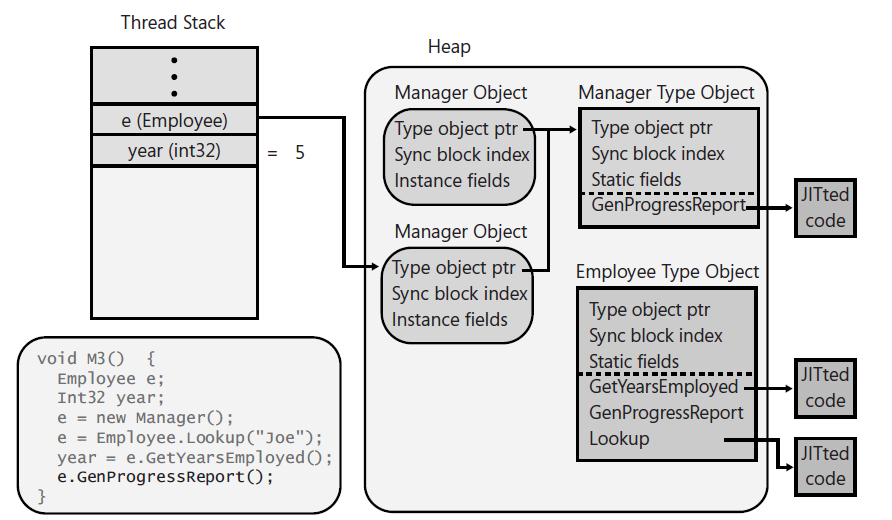
Object.ToString \_ The default implementation returns the full name of the type.

Every object on the heap requires some additional members—called the type object pointer

and the sync block index—used by the CLR to manage the object.

Stack size – 1MB

<https://stackoverflow.com/questions/28656872/why-is-stack-size-in-c-sharp-exactly-1-mb#:~:text=Today's%20PCs%20have%20a%20large,MB%20or%20512%20KB)%3F>



* Calling non-virtual method:  
  When calling a nonvirtual instance method, the JIT compiler locates the type object that corresponds to the type of the variable being used to make the call. In this case, the variable e is defined as an Employee. (If he Employee type didn’t define the method being called, the JIT compiler walks down the class hierarchy toward Object looking for this method. It can do this because each type object has a field in it that refers to its base type; this information is not shown in the figures.)
* Calling static method:  
  When calling a static method, the JIT compiler locates the type object that corresponds to the type that defines the static method. Then, the JIT compiler locates the entry in the type object’s method table that refers to the method being called, JITs the method (if necessary), and calls the JITted code.
* Calling virtual instance method:  
  When calling a virtual instance method, the JIT compiler produces some additional code in the method, which will be executed each time the method is invoked. This code will first look in the variable being used to make the call and then follow the address to the calling object.

All type object are “instances” of System.Type type object (“instance” of itself).

## Primitive types

Any data types the compiler directly supports are called primitive types. Primitive types map directly to types existing in the Framework Class Library (FCL).

Dynamic \_ To the common language runtime (CLR), dynamic is identical to object. However, the C# compiler allows dynamic variables to participate in dynamic dispatch using a simplified syntax.

Cast \_ When performing the arithmetic operation, the first step requires that all operand values be expanded to 32-bit values (or 64-bit values if any operand requires more than 32 bits).

## Reference and Value Types

The generic collection classes allow you to work with collections of value types without requiring that items in the collection be boxed/unboxed.

If the reference doesn’t refer to an object that is a boxed instance of the desired value type, an InvalidCastException is thrown.

### Boxing and Unboxing

#### Boxing

* Memory is allocated from the managed heap. The amount of memory allocated is the size required by the value type’s fields plus the two additional overhead members (the type object pointer and the sync block index) required by all objects on the managed heap.
* The value type’s fields are copied to the newly allocated heap memory.
* The address of the object is returned. This address is now a reference to an object; the value type is now a reference type.

#### Unboxing

Unboxing is not the exact opposite of boxing. The unboxing operation is much less costly than boxing. Unboxing is really just the operation of obtaining a pointer to the raw value type (data fields) contained within an object. In effect, the pointer refers to the unboxed portion in the boxed instance. So, unlike boxing, unboxing doesn’t involve the copying of any bytes in memory. Having made this important clarification, it is important to note that an unboxing operation is typically followed by copying the fields.

## Object Equality and Identity

Object.Equals: default

public class Object {

public virtual Boolean Equals(Object obj) {

// If both references point to the same object,

// they must have the same value.

if (this == obj) return true;

// Assume that the objects do not have the same value.

return false;

}

}

#### Object.Equals override rules

* Equals must be reflexive; that is, x.Equals(x) must return true.
* Equals must be symmetric; that is, x.Equals(y) must return the same value as y.Equals(x).
* Equals must be transitive; that is, if x.Equals(y) returns true and y.Equals(z) returns true, then x.Equals(z) must also return true.
* Equals must be consistent. Provided that there are no changes in the two values being compared, Equals should consistently return true or false.

#### ValueType’s Equals:

1. If the obj argument is null, return false.
2. If the this and obj arguments refer to objects of different types, return false.
3. For each instance field defined by the type, compare the value in the this object with the value in the obj object by calling the field’s Equals method. If any fields are not equal, return false.
4. Return true. Object’s Equals method is not called by ValueType’s Equals method.

#### Few more things:

* implement the System.IEquatable<T> interface’s Equals method
* Overload the == and !=operator methods Usually, you’ll implement these operator methods to internally call the type-safe Equals method
* for the purposes of sorting, implement System.IComparable’s CompareTo method and System.IComparable<T>’s type-safe CompareTo method
* overload the various comparison operator methods (<, <=, >, >=) and implement these methods internally to call the type-safe CompareTo method

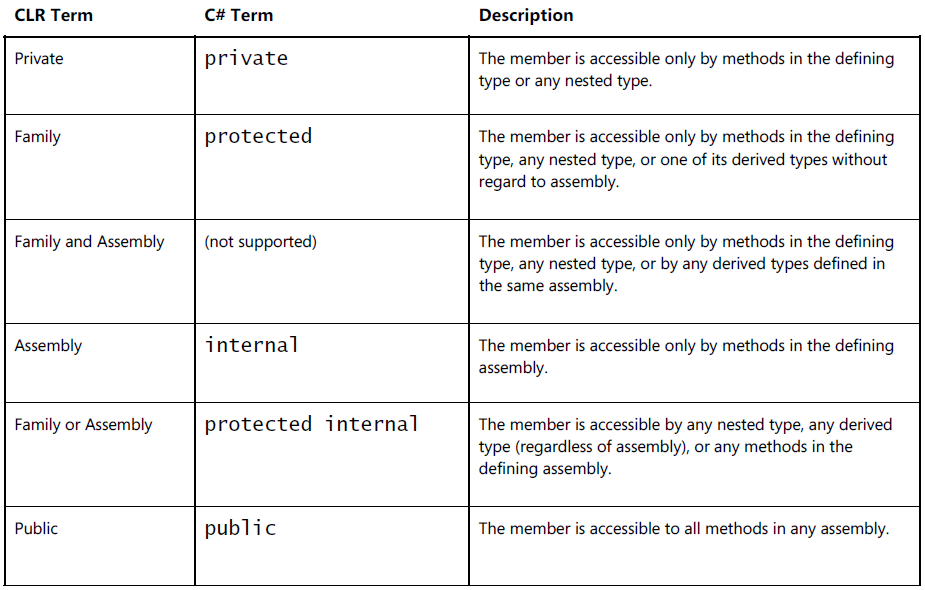
#### Selecting an algorithm for calculating hash codes for instances of your type(Object.GetHashCode):

* Use an algorithm that gives a good random distribution for the best performance of the hash table.
* Your algorithm can also call the base type’s GetHashCode method, including its return value. However, you don’t generally want to call Object’s or ValueType’s GetHashCode method, because the implementation in either method doesn’t lend itself to high-performance hashing algorithms.
* Your algorithm should use at least one instance field.
* Ideally, the fields you use in your algorithm should be immutable; that is, the fields should be initialized when the object is constructed, and they should never again change during the object’s lifetime.
* Your algorithm should execute as quickly as possible.
* Objects with the same value should return the same code. For example, two String objects with the same text should return the same hash code value.

**Operator overloads and Conversion operators** are not part of the

Common Language Specification (CLS).

## Member Accessibility



## Static class

* + The class must be derived directly from System.Object because deriving from any other base class makes no sense since inheritance applies only to objects, and you cannot create an instance of a static class.
  + The class must not implement any interfaces since interface methods are callable only when using an instance of a class.
  + The class must define only static members (fields, methods, properties, and events). Any instance members cause the compiler to generate an error.
  + The class cannot be used as a field, method parameter, or local variable because all of these would indicate a variable that refers to an instance, and this is not allowed. If the compiler detects any of these uses, the compiler issues an error.

## Component Software Programming (CSP)

* + A component (an assembly in .NET) has the feeling of being “published.”
  + A component has an identity (a name, version, culture, and public key).
  + A component forever maintains its identity (the code in an assembly is never statically linked into another assembly; .NET always uses dynamic linking).
  + A component clearly indicates the components it depends upon (reference metadata tables).
  + A component should document its classes and members. C# offers this by allowing in-source Extensible Markup Language (XML) documentation along with the compiler’s /doc command-line switch.
  + A component must specify the security permissions it requires. The CLR’s code access security (CAS) facilities enable this.
  + A component publishes an interface (object model) that won’t change for any servicings. A servicing is a new version of a component whose intention is to be backward compatible with the original version of the component. Typically, a servicing version includes bug fixes, security patches, and possibly some small feature enhancements. But a servicing cannot require any new dependencies or any additional security permissions.

## call vs callvirt

* The call IL instruction can be used to call static, instance, and virtual methods. When the call instruction is used to call a static method, you must specify the type that defines the method that the CLR should call. When the call instruction is used to call an instance or virtual method, you must specify a variable that refers to an object. The call instruction assumes that this variable is not null. In other words, the type of the variable itself indicates which type defines the method that the CLR should call. If the variable’s type doesn’t define the method, base types are checked for a matching method. The call instruction is frequently used to call a virtual method nonvirtually.
* The callvirt IL instruction can be used to call instance and virtual methods, not static methods. When the callvirt instruction is used to call an instance or virtual method, you must specify a variable that refers to an object. When the callvirt IL instruction is used to call a nonvirtual instance method, the type of the variable indicates which type defines the method that the CLR should call. When the callvirt IL instruction is used to call a virtual instance method, the CLR discovers the actual type of the object being used to make the call and then calls the method polymorphically. In order to determine the type, the variable being used to make the call must not be null. In other words, when compiling this call, the JIT compiler generates code that verifies that the variable’s value is not null. If it is null, the callvirt instruction causes the CLR to throw a NullReferenceException. This additional check means that the callvirt IL instruction executes slightly more slowly than the call instruction. Note that this null check is performed even when the callvirt instruction is used to call a nonvirtual instance method.

#### When designing a type, you should try to minimize the number of virtual methods you define.

1. First, calling a virtual method is slower than calling a nonvirtual method.
2. Second, virtual methods cannot be inlined by the JIT compiler, which further hurts performance.
3. Third, virtual methods make versioning of components more brittle, as described in the next section.
4. Fourth, when defining a base type, it is common to offer a set of convenience overloaded methods. If you want these methods to be polymorphic, the best thing to do is to make the most complex method virtual and leave all of the convenience overloaded methods nonvirtual.

## Constructors

In a few situations, an instance of a type can be created without an instance constructor being called. In particular, calling Object’s MemberwiseClone method allocates memory, initializes the object’s overhead fields, and then copies the source object’s bytes to the new object. Also, a constructor is usually not called when deserializing an object with the runtime serializer. The deserialization code allocates memory for the object without calling a constructor using the System.Runtime.Serialization.FormatterServices type's GetUninitializedObject or GetSafeUninitializedObject methods

### Object.MemberwiseClone

The MemberwiseClone method creates a shallow copy by creating a new object, and then copying the nonstatic fields of the current object to the new object. If a field is a value type, a bit-by-bit copy of the field is performed. If a field is a reference type, the reference is copied but the referred object is not; therefore, the original object and its clone refer to the same object.

## Extension methods

#### Rules and Guidelines

* + C# supports extension methods only; it does not offer extension properties, extension events, extension operators, and so on.
  + Extension methods (methods with this before their first argument) must be declared in non-generic, static classes. However, there is no restriction on the name of the class; you can call it whatever you want. Of course, an extension method must have at least one parameter, and only the first parameter can be marked with the this keyword.
  + The C# compiler looks only for extension methods defined in static classes that are themselves defined at the file scope. In other words, if you define the static class nested within another class, the C# compiler will emit the following message: "error CS1109: Extension method must be defined in a top-level static class; StringBuilderExtensions is a nested class."
  + Since the static classes can have any name you want, it takes the C# compiler time to find extension methods as it must look at all the file-scope static classes and scan their static methods for a match. To improve performance and also to avoid considering an extension method that you may not want, the C# compiler requires that you “import” extension methods.
  + It is possible that multiple static classes could define the same extension method. If the compiler detects that two or more extension methods exist, then the compiler issues the following message: "error CS0121: The call is ambiguous between the following methods or properties: 'StringBuilderExtensions.IndexOf(string, char)' and 'AnotherStringBuilderExtensions.IndexOf(string, char)'." To fix this error, you must modify your source code. Specifically, you cannot use the instance method syntax to call this static method anymore; instead you must now use the static method syntax where you explicitly indicate the name of the static class to explicitly tell the compiler which method you want to invoke.
  + You should use this feature sparingly, as not all programmers are familiar with it. For example, when you extend a type with an extension method, you are actually extending derived types with this method as well. Therefore, you should not define an extension method whose first parameter is System.Object, as this method will be callable for all expression types and this will really pollute Visual Studio’s IntelliSense window.
  + There is a potential versioning problem that exists with extension methods. If, in the future, Microsoft adds an IndexOf instance method to their StringBuilder class with the same prototype as my code is attempting to call, then when I recompile my code, the compiler will bind to Microsoft’s IndexOf instance method instead of my static IndexOf method. Because of this, my program will experience different behavior. This versioning problem is another reason why this feature should be used sparingly.

## Partial methods

#### Rules and Guidelines

* + They can only be declared within a partial class or struct.
  + Partial methods must always have a return type of void, and they cannot have any parameters marked with the out modifier. These restrictions are in place because at runtime, the method may not exist and so you can’t initialize a variable to what the method might return because the method might not exist. Similarly, you can’t have an out parameter because the method would have to initialize it and the method might not exist. A partial method may have ref parameters, may be generic, may be instance or static, and may be marked as unsafe.
  + Of course, the defining partial method declaration and the implementing partial method declaration must have identical signatures. If both have custom attributes applied to them, then the compiler combines both methods’ attributes together. Any attributes applied to a parameter are also combined.
  + If there is no implementing partial method declaration, then you cannot have any code that attempts to create a delegate that refers to the partial method. Again, the reason is that the method doesn’t exist at runtime. The compiler produces this message: "error CS0762: Cannot create delegate from method 'Base.OnNameChanging(string)' because it is a partial method without an implementing declaration".
  + Partial methods are always considered to be private methods. However, the C# compiler forbids you from putting the private keyword before the partial method declaration.

## Array

* Array.Clone \_ shallow copy
* Array.Copy \_ The arrays can be reference-type arrays or value-type arrays. Type downcasting is performed, as required.
  + When copying from a reference-type array to a value-type array, each element is unboxed and then copied. When copying from a value-type array to a reference-type array, each element is boxed and then copied.
  + When copying from a reference-type or value-type array to an Object array, an Object is created to hold each value or reference and then copied. When copying from an Object array to a reference-type or value-type array and the assignment is not possible, an InvalidCastException is thrown.
  + If sourceArray and destinationArray are both reference-type arrays or are both arrays of type Object, a shallow copy is performed. A shallow copy of an Array is a new Array containing references to the same elements as the original Array. The elements themselves or anything referenced by the elements are not copied. In contrast, a deep copy of an Array copies the elements and everything directly or indirectly referenced by the elements.
* Array.CopyTo \_shallow copy
* Array.CreateInstance()

## Parameters

### Default Parameters

#### Rules and Guidelines

* You can specify default values for the parameters of methods, constructor methods, and parameterful properties (C# indexers). You can also specify default values for parameters that are part of a delegate definition. Then, when invoking a variable of this delegate type, you can omit the arguments and accept the default values.
* Parameters with default values must come after any parameters that do not have default values. That is, once you define a parameter as having a default value, then all parameters to the right of it must also have default values. For example, in the definition of my M method, I would get a compiler error if I removed the default value ("A") for s. There is one exception to this rule: a params array parameter (discussed later in this chapter) must come after all parameters (including those that have default values), and the array cannot have a default value itself.
* Default values must be constant values known at compile time. This means that you can set default values for parameters of types that C# considers to be primitive types. This also includes enumerated types, and any reference type can be set to null. For a parameter of an arbitrary value type, you can set the default value to be an instance of the value type, with all its fields containing zeroes. You can use the default keyword or the new keyword to express this; both syntaxes produce identical Intermediate Language (IL) code.
* Be careful not to rename parameter variables because any callers who are passing arguments by parameter name will have to modify their code. For example, in the declaration of my M method, if I rename the dt variable to dateTime, then my third call to M in the earlier code will cause the compiler to produce the following message: "error CS1739: The best overload for 'M' does not have a parameter named 'dt'." Be aware that changing a parameter’s default value is potentially dangerous if the method is called from outside the module. A call site embeds the default value into its call. If you later change the parameter’s default value and do not recompile the code containing the call site, then it will call your method passing the old default value. You might want to consider using a default value of 0/null as a sentinel to indicate default behavior; this allows you to change your default without having to recompile all the code with call sites
* You cannot set default values for parameters marked with either the ref or out keywords because there is no way to pass a meaningful default value for these parameters.

### Optional or Named Parameters

#### Rules and Guidelines

* Arguments can be passed in any order; however, named arguments must always appear at the end of the argument list.
* You can pass arguments by name to parameters that do not have default values, but all required arguments must be passed (by position or by name) for the compiler to compile the code.
* C# doesn’t allow you to omit arguments between commas, as in M(1, ,DateTime.Now), because this could lead to unreadable comma-counting code. Pass arguments by way of their parameter name if you want to omit some arguments for parameters with default values.
* To pass an argument by parameter name that requires ref/out, use syntax like this:  
  // Method declaration:  
  private static void M(ref Int32 x) { ... }  
  // Method invocation:  
  Int32 a = 5;  
  M(x: ref a);

# Reflection

**Backus-Naur Form grammar**

#### Use cases

In reality, very few applications will have the need to use the reflection types. Reflection is typically used by class libraries that need to understand a type’s definition in order to provide some rich functionality. For example, the FCL’s serialization mechanism uses reflection to determine what fields a type defines. The serialization formatter can then obtain the values of these fields and write them into a byte stream that is used for sending across the Internet, saving to a file, or copying to the clipboard. Similarly, Visual Studio’s designers use reflection to determine which properties should be shown to developers when laying out controls on their Web Forms or Windows Forms at design time.

Reflection is also used when an application needs to load a specific type from a specific assembly at runtime to accomplish some task. For example, an application might ask the user to provide the name of an assembly and a type. The application could then explicitly load the assembly, construct an instance of the type, and call methods defined in the type. This usage is conceptually similar to calling Win32’s LoadLibrary and GetProcAddress functions. Binding to types and calling methods in this way is frequently referred to as late binding. (Early binding is when the types and methods used by an application are determined at compile time.)

#### Drawbacks

* Reflection prevents type safety at compile time. Since reflection uses strings heavily, you lose type safety at compile time. For example, if you call Type.GetType("int"); to ask reflection to find a type called “int”, the code compiles but returns null at runtime because the CLR knows the “int” type as “System.Int32”.
* Reflection is slow. When using reflection, the names of types and their members are not known at compile time; you discover them at runtime by using a string name to identify each type and member. This means that reflection is constantly performing string searches as the types in the System.Reflection namespace scan through an assembly’s metadata. Often, the string searches are case-insensitive comparisons, which can slow this down even more.

#### If you’re writing an application that will dynamically discover and construct type instances, you should take one of the following approaches:

* + Have the types derive from a base type that is known at compile time. At runtime, construct an instance of the derived type, place the reference in a variable that is of the base type (by way of a cast), and call virtual methods defined by the base type.
  + Have the type implement an interface that is known at compile time. At runtime, construct an instance of the type, place the reference in a variable that is of the interface type (by way of a cast), and call the methods defined by the interface.