Reference and Value types

How types handled in memory

Assignment semantics

Boxing

Types in class hierarchy, object-oriented properties of types

Object class members, how value type implementing them

Types in relation to Garbage Collection

Null-ability

Delegates

Lambda

Closure, local variable capture

Events

Exceptions

Reflection

Types, Method and property info

Attributes

Collections

Interfaces

Iterators and Enumerators

Observable

Arrays

Strings

Search and sort

Hashtable and Dictionary

LINQ

Garbage collection

Generations

Finalization

Leaks

Disposal

GC performance

Forced GC

**How value and reference types handled in memory**

The content of a **value type**variable is simply a value. Value-type instances occupy precisely the memory required to store their fields.

**Reference types** require separate allocations for reference and an instance.

The object consumes as many bytes as its fields, plus additional administrative overhead.

The precise overhead depends on .NET runtime, minimum 8 bytes for **type object pointer** and **sync block index**

**Stack vs Heap storage** The values of value-type variables are stored wherever they are declared.

Stand-alone value types (local variables and method parameters) are usually allocated on the **thread stack**

Value types can be embedded in reference types and allocated on the **heap**,

Value types can be boxed, transferring their storage to the **heap**

|  |
| --- |
| Contrary to popular belief, there isn’t that much of a difference between stacks and heaps in a .NET process. Stacks and heaps are nothing more than ranges of addresses in virtual memory, and there is no inherent advantage in the range of addresses reserved to the stack of a particular thread compared to the range of addresses reserved for the managed heap. Accessing a memory location on the heap is neither faster nor slower than accessing a memory location on the stack.  On the stack, temporal allocation locality (allocations made close together in time) implies spatial locality (storage that is close together in space). In turn, the sequential stack storage tends to perform better with respect to CPU caches and operating system paging systems. Also, memory density on the stack tends to be higher than on the heap because of the reference type overhead. |

**Reference type instances** (objects) are always stored on the heap,

**Static variables** are always stored on the heap,

The reference (pointer) to heap location is stored as a reference variable on the **stack**.

Reference variable can point to location of a reference type instance, or can point nowhere, so value is **null**

Allocation is cheap and fast. Triggering **garbage collection may affect performance**

**Typical sizes of value types**

sizeof(sbyte) sizeof(byte) 1 (8) sizeof(bool) 1 (8)

sizeof(char) 2 (16) sizeof(short) sizeof(ushort) 2 (16) sizeof(int) sizeof(uint) 4 (32)

sizeof(float) sizeof(double) sizeof(long) sizeof(ulong) 8 (64) sizeof(decimal) 16 (128)

**System.Object, System.ValueType, System.Enum**

Every type derives from **System.Object**

Value types do not derive from System.Object directly; they derived from the **System.ValueType** abstract type.

Enums are derived from the **System.Enum** abstract type, which is itself derived from **System.ValueType.**

**Value types are sealed**, which prevents them from being a base type for any other reference or value type.

Cannot have virtual members – because no possibility to override

**Struct** Is a fancy word for value type. Allows declaration of custom value types

Value types don’t have explicit parameterless constructors - to improve the run-time performance.

Main reasons is the cost of creating arrays of a struct type.

Value type can implement one or more **interfaces**.

**Methods of System.Object**

|  |
| --- |
| virtual string ToString()  virtual int GetHashCode()  virtual bool Equals()  virtual void Finalize  Type GetType()  MemberwiseClone  static ReferenceEquals() |

When (JIT) compiler sees that value type overrides the ToString method,

it emits code that calls ToString directly (nonvirtually) without having to do any boxing.

GetHashCode() should be overridden only in conjunction with Equals(). Value must be consistent over life of object.

Also, !=, == operators need to be overridden.

Equals() for reference types compares two reference variables on the stack. Same reference points to **the same** object.

However, calling a **nonvirtual** inherited method (GetType or MemberwiseClone**) always requires boxing** because these methods are defined by System.Object, so the methods expect **this** argument to be a pointer that refers to an object on the heap.

virtual Finalize : virtual method is called when the garbage collector determines that the object is garbage before the memory for the object is reclaimed. Types that require cleanup when collected should override this method.

MemberwiseClone : Nonvirtual method creates a new instance of the type and sets the new object's instance fields to be identical to the **this** object's instance fields. A reference to the new instance is returned.

The type's instance constructor is called **new** returns a reference (or pointer) to the newly created object.

**Equality semantics**

**Boxing** when compiler detects a situation that requires treating a value type instance as a reference type, it emits the box IL. Box is detached from the original value type instance – changes made to one do not affect the other

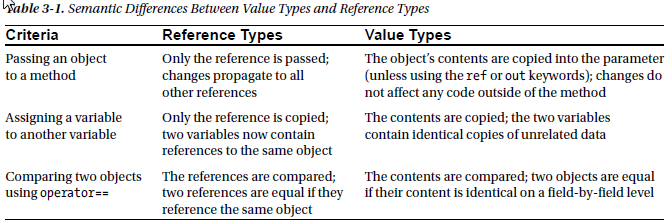
|  |
| --- |
| The JIT compiler has a short-circuit behavior that could permit a direct method call to Equals, because value types are sealed and the virtual dispatch target is determined at compile-time by whether Point2D overrides Equals or not  Whenever the compiler encounters a.Equals(b), it will definitely prefer the second overload to the first, because its parameter type matches more closely the argument type provided. While we’re at it, there are some more methods to overload – often enough, we compare objects using the == and != operators:  Point2D point = ...;  IEquatable<Point2D> equatable = point; //boxing occurs here  However, when making an interface call through a statically typed value type variable, no boxing will occur (this is the same short-circuiting enabled by the constrained IL prefix, discussed above):  Point2D point = ..., anotherPoint = ...;  point.Equals(anotherPoint); //no boxing occurs here, Point2D.Equals(Point2D) is invoked  This is one ground for the common recommendation to make value types immutable, and allow modification only by making more copies. (Consider the System.DateTime API for an example of a well-designed immutable value type.) |

**Assignment semantics**

Objects themselves are *never* passed, either by reference or by value.

Assignment copies a value-type instance

Assigning a reference-type copies the reference, not instance, allows multiple variables to refer to the same object



**Parameter** is declaration and used inside function body, **argument** is passed in call from outside body.

By default all method parameters are passed by value.

Copy of value is created when passing.

Reference is also passed by value. Copy of reference variable is created when passing

CLR allows to pass parameters by reference instead of by value by using the **out** and **ref** keywords

The **out** argument **need not** be assigned before going into the function

The **ref** argument **must** be assigned before it comes *out* of the function

Methods, constructors, and indexers can declare *optional parameters with* *default value*

**Nulability** value type can never be **null;** concept of nullable value types

|  |
| --- |
| public **struct Nullable<T>** where T : **struct**  {  private Boolean hasValue = false; // Assume null  internal T value = default(T); // Assume all bits zero  **Int32**? is a synonym notation for **Nullable<Int32>.** |

Doesn't actually box anything, and returns **null** if nullable instance is **null**

If not **null,** the CLR takes the value out of the nullable instance and boxes it.

**GetType** on a **Nullable<T>** , the CLR actually returns the type **T** instead of **Nullable<T>**

**Types in relation to Garbage Collection**

C# does not allow you to define **Finalize** on value types.

CLR won't call it when a boxed instance is garbage collected.

**Delegates**

Delegate represents **function as object** (of type Delegate), holds function pointer and can invoke.

Assigning a method to a delegate variable creates a delegate *instance*

|  |
| --- |
| **public delegate void Feedback (int value);** // delegate type declaration – **compiler generates new class** :  public class Feedback : System.MulticastDelegate  {  public Feedback(Object object, IntPtr method); // delegate instance constructor taking function pointer  and **this** pointer for calling instance methods (null for static)  **public virtual void Invoke(int value)**; // invoker    public virtual IAsyncResult BeginInvoke(Int32 value, AsyncCallback callback, Object object);  public virtual void EndInvoke(IAsyncResult result);  } |

Delegate has have ***multicast***capability. Delegate instance can reference a **list of target methods**.

.Combine() and Remove() +, += operators combine delegate instances

If one of delegates throws an exception or blocks, it stops all of the subsequent delegates in the chain

MulticastDelegate has GetInvocationList method, so you can call each delegate in a chain explicitly

calling -= on a delegate variable with a single target is equivalent to assigning null to that variable.

Delegates are ***immutable***, calling += or -=, in fact creating a *new* delegate instance.

Caller receives the return value from the last method to be invoked.

Generic delegates are the **Func** and **Action**

Parameter compatibility (**contravariance**). Delegate can have more specific parameter types than its method target.

Return type compatibility (**covariance**) Delegate’s target method may return a more specific type than described by the delegate.

**lambda** expression is a syntax simplification for delegate instantiation

**Closure** is capability to capture local variables from the outer scope

|  |
| --- |
| int **factor** = 2;  BinOp addAndFactor = delegate (int a, int b) { return **factor** \* (a + b); }; |

***Expression trees*** permit anonymous functions to be represented as data structures instead of executable code. Expression trees are values of Expression<D>.

|  |
| --- |
| **Anonymous type** is created through the use of this general form:  new { *nameA* = *valueA*, *nameB* = *valueB*, ... } |
|  |
|  |
|  |

**Events**

**broadcaster** and **subscriber** pattern. The broadcaster is a type that contains a delegate field.

The broadcaster decides when to broadcast by invoking the delegate. A subscriber decides when to start and stop listening by calling += and -= on the broadcaster’s delegate.

|  |
| --- |
| public delegate void PriceChangedHandler (decimal oldPrice, decimal newPrice);  public class Stock  {  public **event** PriceChangedHandler PriceChanged;  public decimal Price  {  get { return price; }  set { PriceChanged (oldPrice, price); // empty, fire event. }  private **EventHandler**<**EventArgs**> NewMail = null; // Private delegate instance field initialized to null  [MethodImpl(MethodImplOptions.Synchronized)]  public void **add\_NewMail**(EventHandler<EventArgs> value) // Public add\_ and remove\_ methods |

When compiler hits event keyword, it wraps delegate and exposes only add/remove methods.

If event keyword omitted, code works the same way, but delegate is public

When a listener registers for an event, the listener is adding an instance of delegate type to the list.

Having non-associated delegate instance as null helps avoiding invocation attempts for 0 listeners

registering event handler creates strong reference **from** event source to listening object. If listener is closed, source still holds reference to listener’s handler delegate instance and listener in not collected

WeekEventManager represents event. IWeakEventListener calls Add(this) to subscribe to event

**D**esign pattern provides dedicated event manager and implements an interface on listeners

|  |
| --- |
| public interface IObservable<**out** T> // typed **covariant**, using the out keyword  {  **IDisposable** Subscribe(IObserver<T> observer);  }  public interface IObserver<**in** T> // typed **contravariant**, using the in keyword  {  void OnNext(T value);  }  using (IEnumerator<T> sourceEnumerator = source.GetEnumerator())  {  while (sourceEnumerator.MoveNext()) { T item = sourceEnumerator.Current; // **Pulls** the data from enumerator }  } // Calls Dispose, telling the source we’re done fetching data.  using (source.Subscribe(item => { /\* Do something with the item \*/ }))  {  // While observing events (source will **Push** data), you can do other stuff.  } // Calls Dispose, **Unsubscribe** - telling the source we’re done receiving events. |

Event handlers for classic events are methods, observers are objects implementing some IObserver<T>

**Exceptions**

Are a structured way to catch and handle errors

The “global” exception handling events for WPF and Windows Forms applications (Application.DispatcherUnhandledException and Application.ThreadException) fire only for exceptions thrown on the main UI thread. You still must handle exceptions on worker threads manually. AppDomain.CurrentDomain.UnhandledException fires on any unhandled exception, but provides no means of preventing the application from shutting down afterward.

Exceptions in Constructor Propagated up the stack to caller. Can be caught there

Exceptions in Finalize Thrown on finalization thread. Will result in termination of process

Exceptions in Catch Propagated up the stack to caller. Can be caught there

Exceptions in Finally Propagated up the stack to caller. Can be caught there

Exceptions in F Thread Will not be caught in main thread try-catch. Results in termination

Exceptions in B Thread Will not be caught in main thread try-catch. Results in termination

Throw ex vs. throw: 'throw;' can only be used in catch block. 'throw e' can be used anywhere.

'throw e' lets you provide a new exception object, such as a wrapper around the original exception.

'throw' can be used to re-throw the original exception within a naked catch block.

‘throw e’ resets parameters on it like the Exception.StackWalk property.

**Collections**



**Collection Interfaces**

IEnumerable<T> indicates that collection has enumera tor and provides it : IEnumerator<T>

ICollection<T> adds size/count, search, add, remove and sync support object SyncRoot {get;}

IList<T> adds indexing support : [index], Insert(i) RemoveAt(i)

IDictionary<K, V>

IComparable defines CompareTo , which determines the sort order.

IEquatable interface defines the Equals

**Iterators and Enumerators**

*IEnumerator* and *IEnumerator(T)* interfaces let you traverse over a collection

IEnumerable<T> indicates that collection has enumera tor and provides it : IEnumerator<T> GetEnumerator();

**Observable Collections**

IBindingList

ObservableCollection<T>

Provides ListChanged event

Supports InotifyCollectionChanged event, InotifyPropertyChanged event

**Arrays**

array represents a fixed number of elements of a particular type.

The elements in an array are stored in a contiguous **block of memory**, providing highly efficient access.

For value types, each element value is allocated as part of the array and assigned to default

For reference type elements the array would have merely allocated number of null references

Because Array is a class, arrays are always (themselves) reference types

Implicitly Derived from System.Array, Implicitly Implement IEnumerable, ICollection, and IList

**Jagged array** can store efficiently many rows of varying lengths. int[][] jagged = new int[3][];

**Strings**

**HashSet**

Is a collection that contains no duplicate elements, and whose elements are in no particular order

**Search and Sort**

The binary search methods are fast, but they work only on sorted arrays and require that the elements be compared for *order* rather than simply *equality*. To this effect, the binary search methods can accept an IComparer or IComparer<T> object to arbitrate on ordering decisions. If no comparer is provided, the type’s default ordering algorithm will be applied, based on its implementation of IComparable / ICompara ble<T>.

The predicate-based searching methods allow a method delegate or lambda expression to arbitrate on whether a given element is a “match.”

**Dictionary and Hashtable**

Collection of key-value pairs

The key is used to determine a unique hash code. Hash code is used to index the data.

Dictionary is generic version of hashtable

Its underlying hashtable works by converting each element’s key into an integer hashcode—a pseudounique value—and then applying an algorithm to convert the hashcode into a hash key. This hash key is used internally to determine which “bucket” an entry belongs to.

If the bucket contains more than one value, a linear search is performed on the bucket. A good hash function does not strive to return strictly unique hashcodes (which would usually be impossible); it strives to return hashcodes that are evenly distributed across the 32-bit integer space. This avoids the scenario of ending up with a few very large (and inefficient) buckets.

A dictionary can work with keys of any type, providing it’s able to determine equality between keys and obtain hashcodes. By default, equality is determined via the key’s object.Equals method, and the pseudounique hashcode hashcode is obtained via the key’s GetHashCode method. This behavior can be changed, either by overriding these methods or by providing an IEqualityComparer object when constructing the dictionary.

**LINQ**

basic units of data in LINQ are *sequences* and *elements*. A sequence is any object that implements IEnumerable<T> and an element is each item in the sequence.

A *query operator* is a method that transforms a sequence. A typical query operator accepts an *input sequence* and emits a transformed *output sequence*. In the Enumera ble class in System.Linq, there are around 40 query operators—all implemented as static extension methods. These are called *standard query operators*.

IEnumerable<string> filteredNames = **names.Where** (n => n.Length >= 4);

Most query operators accept a lambda expression as an argument.

This strategy allows the chaining of query operators. Referred to as *fluent syntax*.

C# also provides another syntax for writing queries, called *query expression s*yntax

Not all query operators return a sequence.

The *element* operators extract one element from the input sequence

The *aggregation* operators return a scalar value; usually of numeric type:

The *quantifiers* return a bool value:

|  |
| --- |
| int firstNumber = numbers.First(); // 10  int lastNumber = numbers.Last(); // 6  int count = numbers.Count(); // 5;  int min = numbers.Min(); // 6;  bool hasTheNumberNine = numbers.Contains (9); // true  bool hasMoreThanZeroElements = numbers.Any(); // true |

An important feature of most query operators is that they **execute not when constructed**, but when *enumerated*

**Garbage Collection**

Static members of types are created when the type is loaded and are considered to be potential roots throughout the entire lifetime of the application domain.

one of the most common .NET **memory leaks** is having a reference to your object **from** a static variable

There are two problems that can arise if the garbage collector executes concurrently with other threads:

• *False negatives*: An object is considered alive even though it is eligible for garbage collection.

• *False positives*: An object is considered dead even though it is still referenced by the application.

Suspending threads for garbage collection is performed at *safe points*. Not every set of two instructions can be interrupted to perform a collection. Suspension occurs when it’s safe to perform the collection, and the CLR tries to suspend threads gracefully

Fundamental property of the garbage collector is its asynchronous background nature.

You can’t predict when a garbage collection will happen, so the lifetime of objects is nondeterministic

**Generations**

Assumes that there is an inherent correlation between the object’s age and its life expectancy.

The objects that have survived a garbage collection in generation 0 are not swept to the beginning of generation 0. Instead, they are *promoted* to generation 1, to reflect the fact that their life expectancy is now longer. As part of this promotion, they are copied from the region of memory occupied by generation 0 to the region of memory occupied by generation

*Generation 1* is the buffer between generation 0 and generation 2. It contains objects that have survived one garbage collection. It is slightly larger than generation 0, but still smaller by several orders of magnitude than the entire available memory space.

Generation 2. there are dynamic thresholds within generation 2 that cause a garbage collection to be triggered, because it does not make sense to wait until the entire memory space is full to

perform a garbage collection.

When a garbage collection occurs within generation 2, it is a full garbage collection.

there’s a fourth space on the heap called the **large object heap** (LOH). It provides space for objects of 85,000 bytes or more. Those are not subject to collection based on generations because they’re deemed too expensive to move around



C is not reachable when there’s no single path through object references that can lead to it from a root. There was an object D holding on to it, but that object by itself is not reachable anymore.

to hand out a raw pointer to a managed object need to the tell the CLR the object should not be moved in memory while something holds on to the pointer. The task of doing this is called pinning

Roots - storage locations to objects on the managed heap.

Roots: global and static object pointers. Local variable/parameter pointers on a thread's stack. CPU registers containing pointers to objects in the heap.

When GC starts, it assumes that all objects in the heap are garbage.

GC starts *marking* phase of the collection, walks up the thread’s stack checking all of the roots. If a root is found to refer to an object, a bit will be turned on in the object’s sync block index field—this is how the object is *marked*.

Once all the roots have been checked, the heap contains marked (reachable) and unmarked (Unreachable) objects.

GC starts compacting phase.

**Finalization**

Any type can override the protected Finalize method defined by System.Object to indicate that it requires automatic finalization. The C# syntax is the ~ File method. This method is called a *finalizer*, and it must be invoked when the object is destroyed.

When an object with a finalizer is created, a reference to it is added to a special runtime-managed queue called the *finalization queue*. This queue is considered a root by the garbage collector, meaning that even if the application has no outstanding reference to the object, it is still kept alive by the finalization queue.

GC detects that the only reference to the object is the reference from the finalization queue. The GC consequently moves the object reference to another runtime-managed queue called the *f-reachable queue*.

The GC signals this event **after** a garbage collection completes, if objects were moved to the f-reachable queue, and as a result the finalizer thread wakes up. The finalizer thread synchronously executes the finalizer.

type that defines a Finalize method should also implement the dispose pattern. type can implement the dispose pattern and not define a Finalize

**Force GC**

To ensure the collection of objects for which collection is delayed by finalizers, you can take the additional step of calling WaitForPendingFinalizers and recollecting:

|  |
| --- |
| GC.Collect();  GC.WaitForPendingFinalizers();  GC.Collect(); |

**GC Performance**

•The greatest risk is temporary objects that creep into generation 2, because this causes frequent full collections.

• Large objects should be long-lived or pooled. A LOH collection is equivalent to a full collection.

• References between generations should be kept to a minimum.

The best practice for finalization is to make it deterministic whenever possible, and delegate the exceptional cases to the non-deterministic finalizer.

Value types embedded in reference types minimize the cost of both phases of garbage collections: if objects are larger, there are less objects to mark, and if objects are larger, the sweep phase copies more memory at each time, which reduces the overhead of copying many small objects around.

Finalizable objects take longer to allocate

Finalizable objects get promoted to older generations

|  |
| --- |
|  |
| **Dispose pattern**  - Only Dispose of resources once  - Prevent the GC from disposing of resources if they've already been manually disposed  a single object can have Dispose or Close called on it multiple times; the first time, the resource should be released, for future calls, the method should just return (no exception should be thrown). It is possible to have multiple threads call Dispose/Close on a single object simultaneously. However, the dispose pattern states that thread synchronization is not required. If you don't know if an object is still in use at a certain point in your code, you should not be calling Dispose/Close  call toGC's **SuppressFinalize** turns on a bit flag associated with the object referred to by its single this parameter. When this flag is on, the CLR knows not to move this object's pointer from the finalization list to the freachable queue  If Dispose is used, it prevents the destructor from being called and manually releases managed and unmanaged resources. If the destructor is called, it only releases unmanaged resources. Managed resources will be de-referenced and (possibly) collected. Using Dispose does not prevent you from continuing to interact with the object! A managed resource may be disposed of, yet still referenced somewhere in the code!  http://www.marcclifton.com/Portals/0/images/disposeFlowChart.jpg  C# language provides a using statement, which offers a simplified syntax for catching exception for result of new (or other create method) and call .Dispose in finally. using statement, you initialize an object and save its reference in a variable. |

**Memory Leaks**