System.Object class

In the .NET universe, every type ultimately derives from a base class named System.Object, which can be represented by the C# ***object*** keyword. The Object class defines a set of common members for every type in the framework. In fact, when you do build a class that does not explicitly define its parent, the compiler automatically derives your type from Object.

Because all classes in .NET are derived from Object, every method defined in the Object class is available in all objects in the system.

- **objA.Equals(object objB)**

```
public virtual bool Equals(object obj);
```

Determines whether the specified object is equal to the current object. It is true if the specified object is equal to the current object; otherwise, false.

The type of comparison between the current instance and the obj parameter depends on whether the current instance is a reference type or a value type.

- If the current instance is a reference type, the Equals(Object) method tests for reference equality, and a call to the Equals(Object) method is equivalent to a call to the ReferenceEquals method. Reference equality means that the object variables that are compared refer to the same object.
- If the current instance is a value type, the Equals(Object) method tests for value equality. Value equality means the following:
  - The two objects are of the same type. As the following example shows, a Byte object that has a value of 12 does not equal an Int32 object that has a value of 12, because the two objects have different run-time types.

```
byte value1 = 12;
int value2 = 12;

object object1 = value1;
object object2 = value2;

Console.WriteLine("{0} ({1}) = {2} ({3}): {4}",
object1, object1.GetType().Name,
object2, object2.GetType().Name,
object1.Equals(object2));

//      The example displays the following output:
//      12 (Byte) = 12 (Int32): False
```

  - The values of the public and private fields of the two objects are equal. The following example tests for value equality. It defines a Person structure, which is a value type, and calls the Person class constructor to instantiate two new Person objects, person1 and person2, which have the same value. As the output from the example shows, although the two object variables refer to different objects, person1 and person2 are equal because they have the same value for the private personName field.

```
public struct Person
{
        private string personName;

        public Person(string name)
        {
```

```
            this.personName = name;
        }
        public override string ToString()
        {
            return this.personName;
        }
    }
    public static void Main()
    {
        Person person1 = new Person("John");
        Person person2 = new Person("John");
        Console.WriteLine("Calling Equals:");
        Console.WriteLine(person1.Equals(person2));
        Console.WriteLine("\nCasting to an Object and calling Equals:");
        Console.WriteLine(((object)person1).Equals((object)person2));
    }
    //    The example displays the following output:
    //    Calling Equals:
    //    True
    //    Casting to an Object and calling Equals:
    //    True
```

When you define your own type, that type inherits the functionality defined by the Equals method of its base type. The following table lists the default implementation of the Equals method for the major categories of types in the .NET Framework.

Type category	Equality defined by	Comments
Class derived directly from Object	Object.Equals(Object)	Reference equality; equivalent to calling Object.ReferenceEquals.
Structure	ValueType.Equals	Value equality; either direct byte-by-byte comparison or field-by-field comparison using reflection.
Enumeration	Enum.Equals	Values must have the same enumeration type and the same underlying value.
Delegate	MulticastDelegate.Equals	Delegates must have the same type with identical invocation lists.
Interface	Object.Equals(Object)	Reference equality.

- **Equals(object objA, object objB)**

public static bool Equals(object objA, object objB);

Determines whether the specified object instances are considered equal. It is true if the objects are considered equal; otherwise, false. If both objAand objB are **null**, the method returns true.

The static Equals(Object, Object) method indicates whether two objects, objA and objB, are equal. It also enables you to test objects whose value is **null** for equality. It compares objA and objB for equality as follows:

- It determines whether the two objects represent the same object reference. If they do, the method returns true. This test is equivalent to calling the ReferenceEquals method. In addition, if both objA and objB are **null**, the method returns true.
- It determines whether either objA or objB is **null**. If so, it returns false.
- If the two objects do not represent the same object reference and neither is **null**, it calls objA.Equals(objB) and returns the result. This means that if objA overrides the Object.Equals(Object) method, this override is called.

- **ReferenceEquals(object objA, object objB)**

public static bool ReferenceEquals(object objA, object objB);

Determines whether the specified Object instances are the same instance. It is true if objA is the same instance as objB or if both are **null**, otherwise, false.

Unlike the Equals method and the equality operator, the ReferenceEquals method cannot be overridden. Because of this, if you want to test two object references for equality and you are unsure about the implementation of the Equals method, you can call the ReferenceEquals method.

However, the return value of the ReferenceEquals method may appear to be anomalous in these two scenarios:

- When comparing value types. If objA and objB are value types, they are boxed before they are passed to the ReferenceEquals method. This means that if both objA and objB represent the same instance of a value type, the ReferenceEquals method nevertheless returns false, as the following example shows.

```
public class Example
{
        public static void Main()
        {
                int int1 = 3;
                Console.WriteLine(Object.ReferenceEquals(int1, int1));
                Console.WriteLine(int1.GetType().IsValueType);
        }
}
//      The example displays the following output:
//      False
//      True
```

- When comparing strings. If objA and objB are strings, the ReferenceEquals method returns true if the string is interned. It does not perform a test for value equality. In the following example, s1 and s2 are equal because they are two instances of a single interned string. However, s3 and s4 are not equal, because although they are have identical string values, that string is not interned.

```
public class Example
{
        public static void Main()

        {
                String s1 = "String1";
                String s2 = "String1";
                Console.WriteLine("s1 = s2: {0}", Object.ReferenceEquals(s1, s2));
                Console.WriteLine("{0} interned: {1}", s1,
                String.IsNullOrEmpty(String.IsInterned(s1)) ? "No" : "Yes");
                String suffix = "A";
                String s3 = "String" + suffix;
                String s4 = "String" + suffix;
                Console.WriteLine("s3 = s4: {0}", Object.ReferenceEquals(s3, s4));
                Console.WriteLine("{0} interned: {1}", s3,
                String.IsNullOrEmpty(String.IsInterned(s3)) ? "No" : "Yes");
        }
}
//      The example displays the following output:
//      s1 = s2: True
//      String1 interned: Yes
//      s3 = s4: False
//      StringA interned: No
```

- **GetHashCode()**

public virtual int GetHashCode();

A hash code is a numeric value (Int32) that is used to insert and identify an object in a hash-based collection such as the Dictionary<TKey,TValue> class, the Hashtable class, or a type derived from the DictionaryBase class. The GetHashCode method provides this hash code for algorithms that need quick checks of object equality.

Two objects that are equal return hash codes that are equal. However, the reverse is not true: equal hash codes do not imply object equality, because different (unequal) objects can have identical hash codes. Furthermore, .NET does not guarantee the default implementation of the GetHashCode method, and the value this method returns may differ between .NET implementations, such as different versions of .NET Framework and .NET Core, and platforms, such as 32-bit and 64-bit platforms. For these reasons, do not use the default implementation of this method as a unique object identifier for hashing purposes.

- **GetType()**

public Type GetType();

Returns the exact runtime type of the current instance.

Because System.Object is the base class for all types in the .NET type system, the GetType method can be used to return Type objects that represent all .NET types. .NET recognizes the following five categories of types:

- Classes, which are derived from System.Object,
- Value types, which are derived from System.ValueType.
- Interfaces, which are derived from System.Object starting with the .NET Framework 2.0.
- Enumerations, which are derived from System.Enum.
- Delegates, which are derived from System.MulticastDelegate.

- **ToString()**

  public virtual string ToString();

  Object.ToString is the major formatting method in the .NET Framework. It converts an object to its string representation so that it is suitable for display. Default implementations of the Object.ToString method return the fully qualified name of the object's type.

  Types frequently override the Object.ToString method to provide a more suitable string representation of a particular type. Types also frequently overload the Object.ToString method to provide support for format strings or culture-sensitive formatting.

- **MemberwiseClone()**

  protected 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.

  For example, consider an object called X that references objects A and B. Object B, in turn, references object C. A shallow copy of X creates new object X2 that also references objects A and B. In contrast, a deep copy of X creates a new object X2 that references the new objects A2 and B2, which are copies of A and B. B2, in turn, references the new object C2, which is a copy of C. The example illustrates the difference between a **shallow** and a **deep** copy operation.

  There are numerous ways to implement a deep copy operation if the shallow copy operation performed by the MemberwiseClone method does not meet your needs. These include the following:

  - Call a class constructor of the object to be copied to create a second object with property values taken from the first object. This assumes that the values of an object are entirely defined by its class constructor.
  - Call the MemberwiseClone method to create a shallow copy of an object, and then assign new objects whose values are the same as the original object to any properties or fields whose values are reference types. The DeepCopy method in the example illustrates this approach.
  - Serialize the object to be deep copied, and then restore the serialized data to a different object variable.
  - Use reflection with recursion to perform the deep copy operation.

- **Finalize()**

  ~Object();

  The Finalize method is used to perform cleanup operations on unmanaged resources held by the current object before the object is destroyed. The method is protected and therefore is accessible only through this class or through a derived class. the Finalize method is called when an object that overrides Finalize is destroyed.

  If a type does override the Finalize method, the garbage collector adds an entry for each instance of the type to an internal structure called the finalization queue. The finalization queue contains entries for all the objects in the managed heap whose finalization code must run before the garbage collector can reclaim their memory. The garbage collector then calls the Finalize method automatically under the following conditions:

  - After the garbage collector has discovered that an object is inaccessible, unless the object has been exempted from finalization by a call to the GC.SuppressFinalize method.
  - **On .NET Framework only**, during shutdown of an application domain, unless the object is exempt from finalization. During shutdown, even objects that are still accessible are finalized.

  Finalize is automatically called only once on a given instance, unless the object is re-registered by using a mechanism such as GC.ReRegisterForFinalize and the GC.SuppressFinalize method has not been subsequently called.

  Finalize operations have the following limitations:

  - The exact time when the finalizer executes is undefined. To ensure deterministic release of resources for instances of your class, implement a Close method or provide a IDisposable.Dispose implementation.
  - The finalizers of two objects are not guaranteed to run in any specific order, even if one object refers to the other. That is, if Object A has a reference to Object B and both have finalizers, Object B might have already been finalized when the finalizer of Object A starts.
  - The thread on which the finalizer runs is unspecified.