System.Object Class

```
[ILASM]
.class public serializable Object

[C#]
public class Object
```

Assembly Info:

- *Name:* mscorlib
- *Public Key:* [00 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00]
- *Version:* 1.0.x.x
- *Attributes:*
    - CLSCompliantAttribute(true)

Summary

Provides support for classes. This class is the root of the object hierarchy.

Library: BCL

Thread Safety: All public static members of this type are safe for multithreaded operations. No instance members are guaranteed to be thread safe.

Description

[*Note:* Classes derived from **System.Object** may override the following methods of the **System.Object** class:

- **System.Object.Equals** - Enables comparisons between objects.

- **System.Object.Finalize** - Performs clean up operations before an object is automatically reclaimed.

- **System.Object.GetHashCode** - Generates a number corresponding to the value of the object (to support the use of a hashtable).

- **System.Object.ToString** - Manufactures a human-readable text string that describes an instance of the class.

]

Object() Constructor

```
[ILASM]
public rtspecialname specialname instance void .ctor()

[C#]
public Object()
```

Summary

Constructs a new instance of the **System.Object** class.

Usage

This constructor is called by constructors in derived classes, but it can also be used to directly create an instance of the **Object** class. This might be useful, for example, if you need to obtain a reference to an object so that you can synchronize on it, as might be the case when using the C# **lock** statement.

Object.Equals(System.Object) Method

```
[ILASM]
.method public hidebysig virtual bool Equals(object obj)

[C#]
public virtual bool Equals(object obj)
```

Summary

Determines whether the specified **System.Object** is equal to the current instance.

Parameters

Parameter	Description
obj	The **System.Object** to compare with the current instance.

Return Value

true if *obj* is equal to the current instance; otherwise, **false**.

Behaviors

The statements listed below are required to be true for all implementations of the **System.Object.Equals** method. In the list, x, y, and z represent non-null object references.

- x.Equals(x) returns **true**.
- x.Equals(y) returns the same value as y.Equals(x).
- (x.Equals(y) && y.Equals(z)) returns **true** if and only if x.Equals(z) returns **true**.
- Successive invocations of x.Equals(y) return the same value as long as the objects referenced by x and y are not modified.
- x.Equals(**null**) returns **false**.

See **System.Object.GetHashCode** for additional required behaviors pertaining to the **System.Object.Equals** method.

[*Note:* Implementations of **System.Object.Equals** should not throw exceptions.]

Default

The **System.Object.Equals** method tests for *referential equality*, which means that **System.Object.Equals** returns **true** if the specified instance of **Object** and the current instance are the same instance; otherwise, it returns **false**.

[*Note:* An implementation of the **System.Object.Equals** method is shown in the following C# code:

```
public virtual bool Equals(Object obj) {


return this == obj;


}
```

]

How and When to Override

For some kinds of objects, it is desirable to have **System.Object.Equals** test for *value equality* instead of referential equality. Such implementations of **Equals** return true if the two objects have the same "value", even if they are not the same instance. The definition of what constitutes an object's "value" is up to the implementer of the type, but it is typically some or all of the data stored in the instance variables of the object. For example, the value of a **System.String** is based on the characters of the string; the **Equals** method of the **System.String** class returns **true** for any two string instances that contain exactly the same characters in the same order.

When the **Equals** method of a base class provides value equality, an override of **Equals** in a class derived from that base class should invoke the inherited implementation of **Equals**.

It is recommended (but not required) that types overriding **System.Object.Equals** also override **System.Object.GetHashCode**. Hashtables cannot be relied on to work correctly if this recommendation is not followed.

If your programming language supports operator overloading, and if you choose to overload the equality operator for a given type, that type should override the **Equals** method. Such implementations of the **Equals** method should return the same results as the equality operator. Following this guideline will help ensure that class library code using **Equals** (such as **System.Collections.ArrayList** and **System.Collections.Hashtable**) behaves in a manner that is consistent with the way the equality operator is used by application code.

If you are implementing a value type, you should follow these guidelines:

- Consider overriding **Equals** to gain increased performance over that provided by the default implementation of **Equals** on **System.ValueType**.

- If you override **Equals** and the language supports operator overloading, you should overload the equality operator for your value type.

For reference types, the guidelines are as follows:

- Consider overriding **Equals** on a reference type if the semantics of the type are based on the fact that the type represents some value(s). For example, reference types such as Point and BigNumber should override **Equals**.

- Most reference types should not overload the equality operator, even if they override **Equals**. However, if you are implementing a reference type that is intended to have value semantics, such as a complex number type, you should override the equality operator.

If you implement **System.IComparable** on a given type, you should override **Equals** on that type.

Usage

The **System.Object.Equals** method is called by methods in collections classes that perform search operations, including the **System.Array.IndexOf** method and the **System.Collections.ArrayList.Contains** method.

Example

Example 1:

The following example contains two calls to the default implementation of **System.Object.Equals**.

[C#]

```
using System;
class MyClass {
   static void Main() {
      Object obj1 = new Object();
      Object obj2 = new Object();
      Console.WriteLine(obj1.Equals(obj2));
      obj1 = obj2;
      Console.WriteLine(obj1.Equals(obj2));
   }
}
```

The output is

```
False


True
```

Example 2:

The following example shows a **Point** class that overrides the **System.Object.Equals** method to provide value equality and a class **Point3D**, which is derived from **Point**. Because Point's override of **System.Object.Equals** is the first in the inheritance chain to introduce value equality, the **Equals** method of the base class (which is inherited from **System.Object** and checks for referential equality) is not invoked. However, **Point3D.Equals** invokes **Point.Equals** because **Point** implements **Equals** in a manner that provides value equality.

[C#]

```
using System;
public class Point: object {
 int x, y;
 public override bool Equals(Object obj) {
 //Check for null and compare run-time types.
 if (obj == null || GetType() != obj.GetType()) return
false;
 Point p = (Point)obj;
 return (x == p.x) && (y == p.y);
 }
 public override int GetHashCode() {
 return x ^ y;
 }
}

class Point3D: Point {
 int z;
 public override bool Equals(Object obj) {
 return base.Equals(obj) && z == ((Point3D)obj).z;
 }
 public override int GetHashCode() {
 return base.GetHashCode() ^ z;
 }
}
```

The **Point.Equals** method checks that the *obj* argument is non-null and that it references an instance of the same type as this object. If either of those checks fail, the method returns false. The **System.Object.Equals** method uses **System.Object.GetType** to determine whether the run-time types of the two objects are identical. (Note that **typeof** is not used here because it returns the static type.) If instead the method had used a check of the form *obj* `is Point`, the check would return true in cases where *obj* is an instance of a subclass of **Point**, even though *obj* and the current instance are not of the same runtime type. Having verified that both objects are of the same type, the method casts *obj* to type **Point** and returns the result of comparing the instance variables of the two objects.

In **Point3D.Equals**, the inherited **Equals** method is invoked before anything else is done; the inherited **Equals** method checks to see that *obj* is non-null, that *obj* is an instance of the same class as this object, and that the inherited instance variables match. Only when the inherited **Equals** returns true does the method compare the instance variables introduced in the subclass. Specifically, the cast to **Point3D** is not executed unless *obj* has been determined to be of type **Point3D** or a subclass of **Point3D**.

Example 3:

In the previous example, operator == (the equality operator) is used to compare the individual instance variables. In some cases, it is appropriate to use the **System.Object.Equals** method to compare instance variables in an **Equals** implementation, as shown in the following example:

[C#]

```
using System;
class Rectangle {
 Point a, b;
 public override bool Equals(Object obj) {
 if (obj == null || GetType() != obj.GetType()) return
false;
 Rectangle r = (Rectangle)obj;
 //Use Equals to compare instance variables
 return a.Equals(r.a) && b.Equals(r.b);
 }
 public override int GetHashCode() {
 return a.GetHashCode() ^ b.GetHashCode();
 }
}
```

Example 4:

In some languages, such as C#, operator overloading is supported. When a type overloads operator ==, it should also override the **System.Object.Equals** method to provide the same functionality. This is typically accomplished by writing the **Equals** method in terms of the overloaded operator ==. For example:

[C#]

```
using System;
public struct Complex {
 double re, im;
 public override bool Equals(Object obj) {
 return obj is Complex && this == (Complex)obj;
 }
 public override int GetHashCode() {
 return re.GetHashCode() ^ im.GetHashCode();
 }
 public static bool operator ==(Complex x, Complex y) {
 return x.re == y.re && x.im == y.im;
 }
 public static bool operator !=(Complex x, Complex y) {
 return !(x == y);
 }
}
```

Because Complex is a C# struct (a value type), it is known that there will be no subclasses of **Complex**. Therefore, the **System.Object.Equals** method need not compare the GetType() results for each object, but can instead use the **is** operator to check the type of the *obj* parameter.

Object.Equals(System.Object, System.Object) Method

```
[ILASM]
.method public hidebysig static bool Equals(object objA,
object objB)

[C#]
public static bool Equals(object objA, object objB)
```

Summary

Determines whether two object references are equal.

Parameters

Parameter	Description
objA	First object to compare.
objB	Second object to compare.

Return Value

true if one or more of the following statements is true:

- *objA* and *objB* refer to the same object,
- *objA* and *objB* are both null references,
- *objA* is not **null** and *objA*.Equals(*objB*) returns true;

otherwise returns **false**.

Description

This static method checks for null references before it calls *objA*.Equals(*objB*) and returns false if either *objA* or *objB* is null. If the Equals(object *obj*) implementation throws an exception, this method throws an exception.

Example

The following example demonstrates the **System.Object.Equals** method.

[C#]

```
using System;

public class MyClass {
   public static void Main() {
   string s1 = "Tom";
   string s2 = "Carol";
   Console.WriteLine("Object.Equals(\"{0}\", \"{1}\") =>
{2}",
      s1, s2, Object.Equals(s1, s2));

   s1 = "Tom";
   s2 = "Tom";
   Console.WriteLine("Object.Equals(\"{0}\", \"{1}\") =>
{2}",
      s1, s2, Object.Equals(s1, s2));

   s1 = null;
   s2 = "Tom";
   Console.WriteLine("Object.Equals(null, \"{1}\") => {2}",
       s1, s2, Object.Equals(s1, s2));

   s1 = "Carol";
   s2 = null;
   Console.WriteLine("Object.Equals(\"{0}\", null) => {2}",
       s1, s2, Object.Equals(s1, s2));

   s1 = null;
   s2 = null;
   Console.WriteLine("Object.Equals(null, null) => {2}",
       s1, s2, Object.Equals(s1, s2));
   }
}
```

The output is

```
Object.Equals("Tom", "Carol") => False


Object.Equals("Tom", "Tom") => True


Object.Equals(null, "Tom") => False
Object.Equals("Carol", null) => False


Object.Equals(null, null) => True
```

Object.Finalize() Method

[ILASM]
```
.method family hidebysig virtual void Finalize()
```

[C#]
```
~Object()
```

Summary

Allows a **System.Object** to perform cleanup operations before the memory allocated for the **System.Object** is automatically reclaimed.

Behaviors

During execution, **System.Object.Finalize** is automatically called after an object becomes inaccessible, unless the object has been exempted from finalization by a call to **System.GC.SuppressFinalize**. During shutdown of an application domain, **System.Object.Finalize** is automatically called on objects that are not exempt from finalization, even those that are still accessible. **System.Object.Finalize** is automatically called only once on a given instance, unless the object is re-registered using a mechanism such as **System.GC.ReRegisterForFinalize** and **System.GC.SuppressFinalize** has not been subsequently called.

Conforming implementations of the CLI are required to make every effort to ensure that for every object that has not been exempted from finalization, the **System.Object.Finalize** method is called after the object becomes inaccessible. However, there may be some circumstances under which **Finalize** is not called. Conforming CLI implementations are required to explicitly specify the conditions under which **Finalize** is not guaranteed to be called. [*Note:* For example, **Finalize** might not be guaranteed to be called in the event of equipment failure, power failure, or other catastrophic system failures.]

In addition to **System.GC.ReRegisterForFinalize** and **System.GC.SuppressFinalize**, conforming implementations of the CLI are allowed to provide other mechanisms that affect the behavior of **System.Object.Finalize**. Any mechanisms provided are required to be specified by the CLI implementation.

The order in which the **Finalize** methods of two objects are run is unspecified, even if one object refers to the other.

The thread on which **Finalize** is run is unspecified.

Every implementation of **System.Object.Finalize** in a derived type is required to call its base type's implementation of **Finalize**. This is the only case in which application code calls **System.Object.Finalize**.

Default

The **System.Object.Finalize** implementation does nothing.

How and When to Override

A type should implement **Finalize** when it uses unmanaged resources such as file handles or database connections that must be released when the managed object that uses them is reclaimed. Because **Finalize** methods may be invoked in any order (including from multiple threads), synchronization may be necessary if the **Finalize** method may interact with other objects, whether accessible or not. Furthermore, since the order in which **Finalize** is called is unspecified, implementers of **Finalize** (or of destructors implemented through overriding Finalize) must take care to correctly handle references to other objects, as their **Finalize** method may already have been invoked. In general, referenced objects should not be considered valid during finalization.

See the **System.IDisposable** interface for an alternate means of disposing of resources.

Usage

For C# developers: Destructors are the C# mechanism for performing cleanup operations. Destructors provide appropriate safeguards, such as automatically calling the base type's destructor. In C# code, **System.Object.Finalize** cannot be called or overridden.

Object.GetHashCode() Method

[ILASM]

```
.method public hidebysig virtual int32 GetHashCode()
```

[C#]

```
public virtual int GetHashCode()
```

Summary

Generates a hash code for the current instance.

Return Value

A **System.Int32** containing the hash code for the current instance.

Description

System.Object.GetHashCode serves as a hash function for a specific type. [*Note:* A hash function is used to quickly generate a number (a hash code) corresponding to the value of an object. Hash functions are used with **hashtables**. A good hash function algorithm rarely generates hash codes that collide. For more information about hash functions, see *The Art of Computer Programming*, Vol. 3, by Donald E. Knuth.]

Behaviors

All implementations of **System.Object.GetHashCode** are required to ensure that for any two object references x and y, if x.Equals(y) == true, then x.GetHashCode() == y.GetHashCode().

Hash codes generated by **System.Object.GetHashCode** need not be unique.

Implementations of **System.Object.GetHashCode** are not permitted to throw exceptions.

Default

The **System.Object.GetHashCode** implementation attempts to produce a unique hash code for every object, but the hash codes generated by this method are not guaranteed to be unique. Therefore, **System.Object.GetHashCode** may generate the same hash code for two different instances.

How and When to Override

It is recommended (but not required) that types overriding **System.Object.GetHashCode** also override **System.Object.Equals**.

Hashtables cannot be relied on to work correctly if this recommendation is not followed.

Usage

Use this method to obtain the hash code of an object. Hash codes should not be persisted (i.e. in a database or file) as they are allowed to change from run to run.

Example

Example 1

In some cases, **System.Object.GetHashCode** is implemented to simply return an integer value. The following example illustrates an implementation of **System.Int32.GetHashCode**, which returns an integer value:

[C#]

```
using System;
public struct Int32 {
 int value;
 //other methods...

 public override int GetHashCode() {
 return value;
 }
}
```

Example 2

Frequently, a type has multiple data members that can participate in generating the hash code. One way to generate a hash code is to combine these fields using an xor (exclusive or) operation, as shown in the following example:

[C#]

```
using System;
public struct Point {
 int x;
 int y;
 //other methods
 public override int GetHashCode() {
 return x ^ y;
 }
}
```

Example 3

The following example illustrates another case where the type's fields are combined using xor (exclusive or) to generate the hash code. Notice that in this example, the fields represent user-defined types, each of which implements **System.Object.GetHashCode** (and should implement **System.Object.Equals** as well):

[C#]

```
using System;
public class SomeType {
 public override int GetHashCode() {
 return 0;
 }
}

public class AnotherType {
 public override int GetHashCode() {
 return 1;
 }
}

public class LastType {
 public override int GetHashCode() {
 return 2;
 }
}
public class MyClass {
 SomeType a = new SomeType();
 AnotherType b = new AnotherType();
 LastType c = new LastType();

 public override int GetHashCode () {
 return a.GetHashCode() ^ b.GetHashCode() ^
c.GetHashCode();
 }
}
```

Avoid implementing **System.Object.GetHashCode** in a manner that results in circular references. In other words, if AClass.GetHashCode calls BClass.GetHashCode, it should not be the case that BClass.GetHashCode calls AClass.GetHashCode.

Example 4

In some cases, the data member of the class in which you are implementing **System.Object.GetHashCode** is bigger than a **System.Int32**. In such cases, you could combine the high order bits of the value with the low order bits using an XOR operation, as shown in the following example:

[C#]

```
using System;
public struct Int64 {
 long value;
 //other methods...

 public override int GetHashCode() {
 return ((int)value ^ (int)(value >> 32));
 }
}
```

Object.GetType() Method

[ILASM]

```
.method public hidebysig instance class System.Type
GetType()
```

[C#]

```
public Type GetType()
```

Summary

Gets the type of the current instance.

Return Value

The instance of **System.Type** that represents the run-time type (the exact type) of the current instance.

Description

For two objects x and y that have identical run-time types, **System.Object.ReferenceEquals**(x.GetType(),y.GetType()) returns **true**.

Example

The following example demonstrates the fact that **System.Object.GetType** returns the run-time type of the current instance:

[C#]

```
using System;
public class MyBaseClass: Object {
}
public class MyDerivedClass: MyBaseClass {
}
public class Test {
   public static void Main() {
   MyBaseClass myBase = new MyBaseClass();
   MyDerivedClass myDerived = new MyDerivedClass();

   object o = myDerived;
   MyBaseClass b = myDerived;

   Console.WriteLine("mybase: Type is {0}",
myBase.GetType());
   Console.WriteLine("myDerived: Type is {0}",
myDerived.GetType());
```

```
   Console.WriteLine("object o = myDerived: Type is {0}",
o.GetType());
   Console.WriteLine("MyBaseClass b = myDerived: Type is
{0}", b.GetType());
   }
}
```

The output is

```
mybase: Type is MyBaseClass


myDerived: Type is MyDerivedClass


object o = myDerived: Type is MyDerivedClass


MyBaseClass b = myDerived: Type is MyDerivedClass
```

Object.MemberwiseClone() Method

```
[ILASM]
.method family hidebysig instance object MemberwiseClone()

[C#]
protected object MemberwiseClone()
```

Summary

Creates a shallow copy of the current instance.

Return Value

A shallow copy of the current instance. The run-time type (the exact type) of the returned object is the same as the run-time type of the object that was copied.

Description

System.Object.MemberwiseClone creates a new instance of the same type as the current instance and then copies each of the object's non-static fields in a manner that depends on whether the field is a value type or a reference type. If the field is a value type, a bit-by-bit copy of all the field's bits is performed. If the field is a reference type, only the reference is copied. The algorithm for performing a shallow copy is as follows (in pseudo-code):

```
for each instance field f in this instance


if (f is a value type)


bitwise copy the field


if (f is a reference type)


copy the reference


end for loop
```

[*Note:* This mechanism is referred to as a shallow copy because it copies rather than clones the non-static fields.]

Because **System.Object.MemberwiseClone** implements the above algorithm, for any object, a, the following statements are required to be true:

- a.MemberwiseClone() is not identical to a.
- a.MemberwiseClone().GetType() is identical to a.GetType().

System.Object.MemberwiseClone does not call any of the type's constructors.

[*Note:* If **System.Object.Equals** has been overridden, a.MemberwiseClone().Equals(a) might return **false**.]

Usage

For an alternate copying mechanism, see **System.ICloneable**.

System.Object.MemberwiseClone is protected (rather than public) to ensure that from verifiable code it is only possible to clone objects of the same class as the one performing the operation (or one of its subclasses). Although cloning an object does not directly open security holes, it does allow an object to be created without running any of its constructors. Since these constructors may establish important invariants, objects created by cloning may not have these invariants established, and this may lead to incorrect program behavior. For example, a constructor might add the new object to a linked list of all objects of this class, and cloning the object would not add the new object to that list -- thus operations that relied on the list to locate all instances would fail to notice the cloned object. By making the method protected, only objects of the same class (or a subclass) can produce a clone and implementers of those classes are (presumably) aware of the appropriate invariants and can arrange for them to be true without necessarily calling a constructor.

Example

The following example shows a class called **MyClass** as well as a representation of the instance of **MyClass** returned by **System.Object.MemberwiseClone**.

[C#]

```
using System;
class MyBaseClass {
   public static string CompanyName = "My Company";
   public int age;
   public string name;
}

class MyDerivedClass: MyBaseClass {

   static void Main() {

   //Create an instance of MyDerivedClass
   //and assign values to its fields.
   MyDerivedClass m1 = new MyDerivedClass();
   m1.age = 42;
   m1.name = "Sam";

   //Do a shallow copy of m1
   //and assign it to m2.
   MyDerivedClass m2 = (MyDerivedClass)
m1.MemberwiseClone();
   }
}
```

A graphical representation of m1 and m2 might look like this

```

+---------------+

|     42        |                           m1

+---------------+

|     +---------|-----------------> "Sam"

+---------------+                    /|\

                                      |

+---------------+                     |

|     42        |                     |      m2

+---------------+                     |

|      +--------|---------------------|

+---------------+
```

Object.ReferenceEquals(System.Object, System.Object) Method

[ILASM]

```
.method public hidebysig static bool ReferenceEquals(object
objA, object objB)
```

[C#]

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

Summary

Determines whether two object references are identical.

Parameters

Parameter	Description
objA	First object to compare.
objB	Second object to compare.

Return Value

True if *a* and *b* refer to the same object or are both null references; otherwise, **false**.

Description

This static method provides a way to compare two objects for reference equality. It does not call any user-defined code, including overrides of **System.Object.Equals**.

Example

[C#]

```
using System;
class MyClass {
   static void Main() {
   object o = null;
   object p = null;
   object q = new Object();
   Console.WriteLine(Object.ReferenceEquals(o, p));
   p = q;
   Console.WriteLine(Object.ReferenceEquals(p, q));
   Console.WriteLine(Object.ReferenceEquals(o, p));
```

```
    }
}
```

The output is

```
True


True


False
```

Object.ToString() Method

[ILASM]

```
.method public hidebysig virtual string ToString()
```

[C#]

```
public virtual string ToString()
```

Summary

Creates and returns a **System.String** representation of the current instance.

Return Value

A **System.String** representation of the current instance.

Behaviors

System.Object.ToString returns a string whose content is intended to be understood by humans. Where the object contains culture-sensitive data, the string representation returned by **System.Object.ToString** takes into account the current system culture. For example, for an instance of the **System.Double** class whose value is zero, the implementation of **System.Double.ToString** might return "0.00" or "0,00" depending on the current UI culture. [*Note:* Although there are no exact requirements for the format of the returned string, it should as much as possible reflect the value of the object as perceived by the user.]

Default

System.Object.ToString is equivalent to calling **System.Object.GetType** to obtain the **System.Type** object for the current instance and then returning the result of calling the **System.Object.ToString** implementation for that type. [*Note:* The value returned includes the full name of the type.]

How and When to Override

It is recommended, but not required, that **System.Object.ToString** be overridden in a derived class to return values that are meaningful for that type. For example, the base data types, such as **System.Int32**, implement **System.Object.ToString** so that it returns the string form of the value the object represents.

Subclasses that require more control over the formatting of strings than **System.Object.ToString** provides should implement **System.IFormattable**, whose **System.Object.ToString** method uses the culture of the current thread.

Example

The following example outputs the textual description of the value of an object of type **System.Object** to the console.

[C#]

```
using System;

class MyClass {
   static void Main() {
      object o = new object();
      Console.WriteLine (o.ToString());
   }
}
```

The output is

```
System.Object
```