

1 System.Object Class

```
2 [ILAsm]  
3 .class public serializable Object  
  
4 [C#]  
5 public class Object
```

6 Assembly Info:

- 7 • *Name:* mscorlib
- 8 • *Public Key:* [00 00 00 00 00 00 00 00 04 00 00 00 00 00 00]
- 9 • *Version:* 2.0.x.x
- 10 • *Attributes:*
 - 11 ○ CLSCompliantAttribute(true)

12 Summary

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

14 **Library:** BCL

15

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

18

19 Description

20 [Note: Classes derived from System.Object can override the following methods of the
21 System.Object class:

- 22 • System.Object.Equals - Enables comparisons between objects.
- 23 • System.Object.Finalize - Performs clean up operations before an object is
24 automatically reclaimed.
- 25 • System.Object.GetHashCode - Generates a number corresponding to the value of
26 the object (to support the use of a hashtable).
- 27 • System.Object.ToString - Manufactures a human-readable text string that
28 describes an instance of the class.

29]

30

1 Object() Constructor

```
2 [ILAsm]  
3 public rtspecialname specialname instance void .ctor()  
4 [C#]  
5 public Object()
```

6 Summary

7 Constructs a new instance of the `System.Object` class.

8 Usage

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

13

14

1 Object.Equals(System.Object) Method

```
2 [ILAsm]  
3 .method public hidebysig virtual bool Equals(object obj)  
4 [C#]  
5 public virtual bool Equals(object obj)
```

6 Summary

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

8 Parameters

Parameter	Description
<i>obj</i>	The <code>System.Object</code> to compare with the current instance.

9 Return Value

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

12 Behaviors

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

- 16 • `x.Equals(x)` returns true.
- 17 • `x.Equals(y)` returns the same value as `y.Equals(x)`.
- 18 • If `(x.Equals(y) && y.Equals(z))` returns true, then `x.Equals(z)` returns true.
- 19 • Successive invocations of `x.Equals(y)` return the same value as long as the objects
20 referenced by *x* and *y* are not modified.
- 21 • `x.Equals(null)` returns false for non-null *x*.

22 See `System.Object.GetHashCode` for additional required behaviors pertaining to the
23 `System.Object.Equals` method.

24
25 [Note: Implementations of `System.Object.Equals` should not throw exceptions.]
26
27

28 Default

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

4
5 [Note: An implementation of the `System.Object.Equals` method is shown in the
6 following C# code:

```
7  
8 public virtual bool Equals(Object obj) {  
9  
10  
11     return this == obj;  
12  
13  
14 }  
15  
16  
17 ]
```

18 How and When to Override

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

27
28 When the `Equals` method of a base class provides value equality, an override of `Equals`
29 in a class derived from that base class should invoke the inherited implementation of
30 `Equals`.

31
32 All implementations of `System.Object.GetHashCode` are required to ensure that for any
33 two object references `x` and `y`, if `x.Equals(y) == true`, then `x.GetHashCode() ==`
34 `y.GetHashCode()`.

35
36 If your programming language supports operator overloading, and if you choose to
37 overload the equality operator for a given type, that type should override the `Equals`
38 method. Such implementations of the `Equals` method should return the same results as
39 the equality operator. Following this guideline will help ensure that class library code
40 using `Equals` (such as `System.Collections.ArrayList` and
41 `System.Collections.Hashtable`) behaves in a manner that is consistent with the way
42 the equality operator is used by application code.

43
44 If you are implementing a value type, you should follow these guidelines:

- 45 • Consider overriding `Equals` to gain increased performance over that provided by the
46 default implementation of `Equals` on `System.ValueType`.
- 47 • If you override `Equals` and the language supports operator overloading, you should
48 overload the equality operator for your value type.

1 For reference types, the guidelines are as follows:

- 2 • Consider overriding `Equals` on a reference type if the semantics of the type are
3 based on the fact that the type represents some value(s). For example, reference
4 types such as `Point` and `BigInteger` should override `Equals`.
- 5 • Most reference types should not overload the equality operator, even if they override
6 `Equals`. However, if you are implementing a reference type that is intended to have
7 value semantics, such as a complex number type, you should override the equality
8 operator.

9 If you implement `System.IComparable` on a given type, you should override `Equals` on that
10 type.

11 Usage

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

15

16 Example

17 Example 1:

18

19 The following example contains two calls to the default implementation of
20 `System.Object.Equals`.

21

22 [C#]

```
23 using System;  
24 class MyClass {  
25     static void Main() {  
26         Object obj1 = new Object();  
27         Object obj2 = new Object();  
28         Console.WriteLine(obj1.Equals(obj2));  
29         obj1 = obj2;  
30         Console.WriteLine(obj1.Equals(obj2));  
31     }  
32 }
```

33 The output is

34

35 False

36

37

38 True

39

40

41 Example 2:

42

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

8
9 [C#]

```
10 using System;
11 public class Point: object {
12     int x, y;
13     public override bool Equals(Object obj) {
14         //Check for null and compare run-time types.
15         if (obj == null || GetType() != obj.GetType()) return false;
16         Point p = (Point)obj;
17         return (x == p.x) && (y == p.y);
18     }
19     public override int GetHashCode() {
20         return x ^ y;
21     }
22 }
23
24 class Point3D: Point {
25     int z;
26     public override bool Equals(Object obj) {
27         return base.Equals(obj) && z == ((Point3D)obj).z;
28     }
29     public override int GetHashCode() {
30         return base.GetHashCode() ^ z;
31     }
32 }
```

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

42

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

49

50 Example 3:

1
2 In the previous example, operator == (the equality operator) is used to compare the
3 individual instance variables. In some cases, it is appropriate to use the
4 System.Object.Equals method to compare instance variables in an Equals
5 implementation, as shown in the following example:

```
6 [C#  
7  
8 using System;  
9 class Rectangle {  
10     Point a, b;  
11     public override bool Equals(Object obj) {  
12         if (obj == null || GetType() != obj.GetType()) return false;  
13         Rectangle r = (Rectangle)obj;  
14         //Use Equals to compare instance variables  
15         return a.Equals(r.a) && b.Equals(r.b);  
16     }  
17     public override int GetHashCode() {  
18         return a.GetHashCode() ^ b.GetHashCode();  
19     }  
20 }
```

21 Example 4:

22
23 In some languages, such as C#, operator overloading is supported. When a type overloads
24 operator ==, it should also override the System.Object.Equals method to provide the
25 same functionality. This is typically accomplished by writing the Equals method in terms of
26 the overloaded operator ==. For example:

```
27 [C#  
28  
29 using System;  
30 public struct Complex {  
31     double re, im;  
32     public override bool Equals(Object obj) {  
33         return obj is Complex && this == (Complex)obj;  
34     }  
35     public override int GetHashCode() {  
36         return re.GetHashCode() ^ im.GetHashCode();  
37     }  
38     public static bool operator ==(Complex x, Complex y) {  
39         return x.re == y.re && x.im == y.im;  
40     }  
41     public static bool operator !=(Complex x, Complex y) {  
42         return !(x == y);  
43     }  
44 }
```

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

49

1 Object.Equals(System.Object, System.Object)

2 Method

```
3 [ILAsm]  
4 .method public hidebysig static bool Equals(object objA, object objB)  
5 [C#]  
6 public static bool Equals(object objA, object objB)
```

7 Summary

8 Determines whether two object references are equal.

9 Parameters

Parameter	Description
<i>objA</i>	First object to compare.
<i>objB</i>	Second object to compare.

10

11 Return Value

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

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

16 otherwise returns false.

17 Description

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

21 Example

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

23
24

```
24 [C#]  
25 using System;  
26
```

```

1 public class MyClass {
2     public static void Main() {
3         string s1 = "Tom";
4         string s2 = "Carol";
5         Console.WriteLine("Object.Equals(\"{0}\", \"{1}\") => {2}",
6             s1, s2, Object.Equals(s1, s2));
7
8         s1 = "Tom";
9         s2 = "Tom";
10        Console.WriteLine("Object.Equals(\"{0}\", \"{1}\") => {2}",
11            s1, s2, Object.Equals(s1, s2));
12
13        s1 = null;
14        s2 = "Tom";
15        Console.WriteLine("Object.Equals(null, \"{1}\") => {2}",
16            s1, s2, Object.Equals(s1, s2));
17
18        s1 = "Carol";
19        s2 = null;
20        Console.WriteLine("Object.Equals(\"{0}\", null) => {2}",
21            s1, s2, Object.Equals(s1, s2));
22
23        s1 = null;
24        s2 = null;
25        Console.WriteLine("Object.Equals(null, null) => {2}",
26            s1, s2, Object.Equals(s1, s2));
27    }
28 }

```

29 The output is

```

31
32 Object.Equals("Tom", "Carol") => False
33
34
35 Object.Equals("Tom", "Tom") => True
36
37
38 Object.Equals(null, "Tom") => False
39
40
41 Object.Equals("Carol", null) => False
42
43
44 Object.Equals(null, null) => True
45

```

46

1 Object.Finalize() Method

```
2 [ILAsm]  
3 .method family hidebysig virtual void Finalize()  
4 [C#]  
5 ~Object()
```

6 Summary

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

9 Behaviors

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

18
19 Conforming implementations of the CLI are required to make every effort to ensure that
20 for every object that has not been exempted from finalization, the
21 `System.Object.Finalize` method is called after the object becomes inaccessible.
22 However, there might be some circumstances under which `Finalize` is not called.
23 Conforming CLI implementations are required to explicitly specify the conditions under
24 which `Finalize` is not guaranteed to be called. [*Note:* For example, `Finalize` might not
25 be guaranteed to be called in the event of equipment failure, power failure, or other
26 catastrophic system failures.]

27
28
29
30 In addition to `System.GC.ReRegisterForFinalize` and `System.GC.SuppressFinalize`,
31 conforming implementations of the CLI are allowed to provide other mechanisms that
32 affect the behavior of `System.Object.Finalize`. Any mechanisms provided are required
33 to be specified by the CLI implementation.

34
35 The order in which the `Finalize` methods of two objects are run is unspecified, even if
36 one object refers to the other.

37
38 The thread on which `Finalize` is run is unspecified.

39
40 Every implementation of `System.Object.Finalize` in a derived type is required to call
41 its base type's implementation of `Finalize`. This is the only case in which application
42 code calls `System.Object.Finalize`.

43 Default

1 The `System.Object.Finalize` implementation does nothing.

2 **How and When to Override**

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

13
14 See the `System.IDisposable` interface for an alternate means of disposing of resources.

15 **Usage**

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

20

21

1 Object.GetHashCode() Method

```
2 [ILAsm]  
3 .method public hidebysig virtual int32 GetHashCode()  
4 [C#]  
5 public virtual int GetHashCode()
```

6 Summary

7 Generates a hash code for the current instance.

8 Return Value

9 A `System.Int32` containing the hash code for the current instance.

10 Description

11 `System.Object.GetHashCode` serves as a hash function for a specific type. [Note: A
12 hash function is used to quickly generate a number (a hash code) corresponding to the
13 value of an object. Hash functions are used with `hashtables`. A good hash function
14 algorithm rarely generates hash codes that collide. For more information about hash
15 functions, see *The Art of Computer Programming*, Vol. 3, by Donald E. Knuth.]
16
17

18 Behaviors

19 All implementations of `System.Object.GetHashCode` are required to ensure that for any
20 two object references `x` and `y`, if `x.Equals(y) == true`, then `x.GetHashCode() ==`
21 `y.GetHashCode()`.
22

23 Hash codes generated by `System.Object.GetHashCode` need not be unique.
24

25 Implementations of `System.Object.GetHashCode` are not permitted to throw
26 exceptions.

27 Default

28 The `System.Object.GetHashCode` implementation attempts to produce a unique hash
29 code for every object, but the hash codes generated by this method are not guaranteed
30 to be unique. Therefore, `System.Object.GetHashCode` can generate the same hash
31 code for two different instances.
32

33 How and When to Override

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

4

5 Usage

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

8

9 Example

10 Example 1

11

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

15

16 [C#]

```
17 using System;  
18 public struct Int32 {  
19     int value;  
20     //other methods...  
21  
22     public override int GetHashCode() {  
23         return value;  
24     }  
25 }
```

26 Example 2

27

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

31

32 [C#]

```
33 using System;  
34 public struct Point {  
35     int x;  
36     int y;  
37     //other methods  
38  
39     public override int GetHashCode() {  
40         return x ^ y;  
41     }  
42 }
```

43 Example 3

44

45 The following example illustrates another case where the type's fields are combined using

1 xor (exclusive or) to generate the hash code. Notice that in this example, the fields
2 represent user-defined types, each of which implements `System.Object.GetHashCode` (and
3 should implement `System.Object.Equals` as well):

```
4  
5 [C#]  
6 using System;  
7 public class SomeType {  
8     public override int GetHashCode() {  
9         return 0;  
10    }  
11 }  
12  
13 public class AnotherType {  
14     public override int GetHashCode() {  
15         return 1;  
16     }  
17 }  
18  
19 public class LastType {  
20     public override int GetHashCode() {  
21         return 2;  
22     }  
23 }  
24 public class MyClass {  
25     SomeType a = new SomeType();  
26     AnotherType b = new AnotherType();  
27     LastType c = new LastType();  
28  
29     public override int GetHashCode () {  
30         return a.GetHashCode() ^ b.GetHashCode() ^ c.GetHashCode();  
31     }  
32 }
```

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

36
37 Example 4

38
39 In some cases, the data member of the class in which you are implementing
40 `System.Object.GetHashCode` is bigger than a `System.Int32`. In such cases, you could
41 combine the high order bits of the value with the low order bits using an XOR operation, as
42 shown in the following example:

```
43  
44 [C#]  
45 using System;  
46 public struct Int64 {  
47     long value;  
48     //other methods...  
49  
50     public override int GetHashCode() {  
51         return ((int)value ^ (int)(value >> 32));  
52     }  
53 }
```

1 }
2

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
```

```
1
2
3 myDerived: Type is MyDerivedClass
4
5
6 object o = myDerived: Type is MyDerivedClass
7
8
9 MyBaseClass b = myDerived: Type is MyDerivedClass
10
11
```

1 Object.MemberwiseClone() Method

```
2 [ILAsm]  
3 .method family hidebysig instance object MemberwiseClone()  
4 [C#]  
5 protected object MemberwiseClone()
```

6 Summary

7 Creates a shallow copy of the current instance.

8 Return Value

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

11 Description

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

18 for each instance field f in this instance

21 if (f is a value type)

25 bitwise copy the field

28 if (f is a reference type)

31 copy the reference

34 end for loop

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

41
42 Because System.Object.MemberwiseClone implements the above algorithm, for any
43 object, a, the following statements are required to be true:

- 1 • a.MemberwiseClone() is not identical to a.
- 2 • a.MemberwiseClone().GetType() is identical to a.GetType().

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

4
5 [Note: If System.Object.Equals has been overridden, a.MemberwiseClone().Equals(a)
6 might return false.]

7
8

9 Usage

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

11
12 System.Object.MemberwiseClone is protected (rather than public) to ensure that from
13 verifiable code it is only possible to clone objects of the same class as the one
14 performing the operation (or one of its subclasses). Although cloning an object does not
15 directly open security holes, it does allow an object to be created without running any of
16 its constructors. Since these constructors might establish important invariants, objects
17 created by cloning might not have these invariants established, and this can lead to
18 incorrect program behavior. For example, a constructor might add the new object to a
19 linked list of all objects of this class, and cloning the object would not add the new
20 object to that list -- thus operations that relied on the list to locate all instances would
21 fail to notice the cloned object. By making the method protected, only objects of the
22 same class (or a subclass) can produce a clone and implementers of those classes are
23 (presumably) aware of the appropriate invariants and can arrange for them to be true
24 without necessarily calling a constructor.

25 Example

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

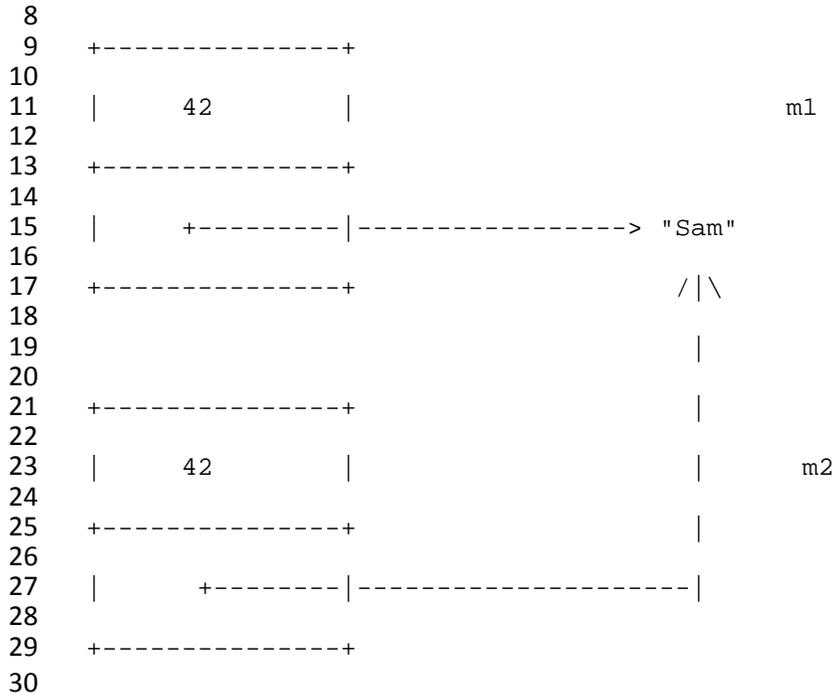
```
28 [C#]
29
30 using System;
31 class MyBaseClass {
32     public static string CompanyName = "My Company";
33     public int age;
34     public string name;
35 }
36
37 class MyDerivedClass: MyBaseClass {
38
39     static void Main() {
40
41         //Create an instance of MyDerivedClass
42         //and assign values to its fields.
43         MyDerivedClass m1 = new MyDerivedClass();
44         m1.age = 42;
45         m1.name = "Sam";
```

```

1
2 //Do a shallow copy of m1
3 //and assign it to m2.
4 MyDerivedClass m2 = (MyDerivedClass) m1.MemberwiseClone();
5 }
6 }

```

7 A graphical representation of m1 and m2 might look like this



1 Object.ReferenceEquals(System.Object, 2 System.Object) Method

```
3 [ILAsm]  
4 .method public hidebysig static bool ReferenceEquals(object objA, object  
5 objB)  
  
6 [C#]  
7 public static bool ReferenceEquals(object objA, object objB)
```

8 Summary

9 Determines whether two object references are identical.

10 Parameters

Parameter	Description
<i>objA</i>	First object to compare.
<i>objB</i>	Second object to compare.

11

12 Return Value

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

14 Description

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

17 Example

```
18 [C#]  
  
19 using System;  
20 class MyClass {  
21     static void Main() {  
22         object o = null;  
23         object p = null;  
24         object q = new Object();  
25         Console.WriteLine(Object.ReferenceEquals(o, p));  
26         p = q;  
27         Console.WriteLine(Object.ReferenceEquals(p, q));  
28         Console.WriteLine(Object.ReferenceEquals(o, p));  
29     }  
30 }
```

```
1
2 The output is
3
4 True
5
6
7 True
8
9
10 False
11
12
```

1 Object.ToString() Method

```
2 [ILAsm]  
3 .method public hidebysig virtual string ToString()  
4 [C#]  
5 public virtual string ToString()
```

6 Summary

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

8 Return Value

9 A `System.String` representation of the current instance.

10 Behaviors

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

19

20 Default

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

25

26 How and When to Override

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

31

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

35 Example

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

3
4 [C#]

```
5 using System;  
6  
7 class MyClass {  
8     static void Main() {  
9         object o = new object();  
10        Console.WriteLine (o.ToString());  
11    }  
12 }  
13
```

14 The output is

15
16 System.Object

17