C# Object-Oriented Programming Concepts - Deep Dive

The Object Class - Universal Base Type

What is the Object Class?

In C#, **System.Object** (or simply `object`) is the ultimate base class for all types in the .NET type system. Every single type in C#, whether it's a class, struct, interface, enum, or delegate, implicitly inherits from `Object`.

Key Characteristics

Universal Parent: All types derive from Object, making it the root of the type hierarchy.

Four Virtual Methods: The Object class provides four key virtual methods that can be overridden:

- `ToString()`
- `Equals(object obj)`
- `GetHashCode()`
- `GetType()` (sealed, cannot be overridden)

Type Flexibility: Because everything inherits from Object, you can create variables of type `object` that can hold any value:

```
object obj = new Employee();  // Holds a reference type
obj = 1;                      // Holds a value type (int)
obj = new Complex();          // Holds a struct
obj = "ali";                  // Holds a string
```

Why Does This Matter?

Polymorphism: Enables you to write generic code that works with any type.

Collections: Early .NET collections (like ArrayList) used object to store any type before generics were introduced.

Flexibility: Allows creating heterogeneous collections:

```
object[] arr = new object[4];
arr[0] = 1;                // int
arr[1] = new Employee(); // Employee object
```

```
arr[2] = "text";          // string
arr[3] = 3.14;            // double
```

Limitations of Using Object

Loss of Type Information: When you store something in an object variable, you lose compile-time type checking.

No IntelliSense: You can't access type-specific members without casting:

```
object o1 = 2;
// o1.  <-- IntelliSense won't show int methods
```

Performance Overhead: Value types must be boxed (converted to reference types) when assigned to object, which has performance implications.

Struct vs Class

Fundamental Differences

Memory Allocation

Struct (Value Type):

- Allocated on the **stack** (in most cases)
- Contains the actual data directly
- Fast allocation and deallocation
- Automatic memory cleanup when out of scope

Class (Reference Type):

- Allocated on the **heap**
- The variable contains a reference (pointer) to the data
- Managed by garbage collector
- More overhead but more flexible

The Complex Struct Example

```
struct Complex
{
    public int real { get; set; }
```

```
    public int img { get; set; }

    public override string ToString()
    {
        return $"{real}+{img}i";
    }
}
```

Value Semantics vs Reference Semantics

Structs have value semantics:

```
Complex c1 = new Complex() { real = 1, img = 2 };
Complex c2 = c1;  // COPIES the entire struct
c2.real = 5;      // Only c2 is modified, c1 remains unchanged
```

Classes have reference semantics:

```
Employee em1 = new Employee() { id = 1, name = "Ali" };
Employee em2 = em1;  // COPIES the reference, both point to same object
em2.name = "Ahmed";  // Both em1 and em2 reflect this change
```

When to Use Struct vs Class

Use Struct When:

- The type represents a single value (like a point, coordinate, or complex number)
- Size is small (Microsoft recommends under 16 bytes)
- Type is immutable (values don't change after creation)
- You don't need inheritance
- You need value semantics (independent copies)

Use Class When:

- The type represents a complex entity with behavior
- You need inheritance or polymorphism
- Size is large (structs are copied entirely on assignment)
- You need reference semantics (multiple variables pointing to same object)
- The type has a long lifetime

Important Struct Rules

No Parameterless Constructor: Structs cannot have explicit parameterless constructors (before C# 10).

All Fields Must Be Initialized: The default constructor initializes all fields to their default values.

Cannot Inherit: Structs cannot inherit from other structs or classes (but can implement interfaces).

Sealed by Default: Structs are implicitly sealed.

Boxing Warning: Structs inherit from `System.ValueType`, which inherits from `System.Object`, so they can be boxed.

Properties in C#

What Are Properties?

Properties are members that provide a flexible mechanism to read, write, or compute the values of private fields. They use accessor methods (get and set) but are accessed like fields.

Auto-Implemented Properties

In your code, both `Complex` and `Employee` use auto-implemented properties:

```
public int real { get; set; }
public string name { get; set; }
```

Behind the Scenes: The compiler automatically creates a private backing field and simple get/set accessors.

Why Use Properties Instead of Public Fields?

Encapsulation: You can add validation or logic later without breaking existing code.

Flexibility: Can make properties read-only, write-only, or computed.

Versioning: Properties provide a stable interface even if internal implementation changes.

Debugging: Can set breakpoints in property accessors.

Data Binding: Many frameworks (like WPF, ASP.NET) work better with properties than fields.

Property Variations

Read-Only Property:

```
public int Age { get; }  // Can only be set in constructor
```

Computed Property:

```
public string FullName
{
    get { return $"{FirstName} {LastName}"; }
}
```

Property with Validation:

```
private int _age;
public int Age
{
    get { return _age; }
    set
    {
        if (value >= 0 && value <= 150)
            _age = value;
        else
            throw new ArgumentException("Invalid age");
    }
}
```

Init-Only Property (C# 9+):

```
public int Id { get; init; }  // Can only be set during object initialization
```

ToString Method Override

Purpose of ToString

The `ToString()` method converts an object to its string representation. Every object in .NET has this method because it's defined in the `Object` class.

Default Behavior

Without overriding, `ToString()` returns the fully qualified name of the type:

```
Employee em = new Employee();
Console.WriteLine(em.ToString());
// Output: Day4PII.Employee (namespace.typename)
```

Custom Override

Both your `Complex` struct and `Employee` class override `ToString()`:

```
public override string ToString()
{
    return $"{real}+{img}i";  // Returns "4+2i" format
}

public override string ToString()
{
    return $"{id}-{name}-{age} years old";  // Returns "1-ahmed-20 years old"
}
```

Why Override ToString?

Debugging: Makes debugging much easier when inspecting objects.

Logging: Provides meaningful output in log files.

Display: Useful for displaying object state to users.

String Interpolation: Works seamlessly with string interpolation and concatenation.

ToString Best Practices

Keep It Simple: Should be quick and not throw exceptions.

Human Readable: Format for human consumption, not for parsing.

Avoid Side Effects: Should not modify the object state.

Be Consistent: All objects of the same type should follow the same format.

Implicit Calls to ToString

ToString is called automatically in many scenarios:

```
Employee em = new Employee() { id = 1, name = "ahmed", age = 20 };

Console.WriteLine(em);           // Implicit call
```

```
string s = "Employee: " + em;     // Implicit call
string s2 = $"Data: {em}";        // Implicit call in interpolation
```

Equals Method Override

Purpose of Equals

The `Equals()` method determines whether two objects are considered equal. By default, for reference types, it checks if two references point to the same object (reference equality).

Default Reference Equality (Classes)

Without overriding:

```
Employee em1 = new Employee() { id = 1, name = "ali", age = 20 };
Employee em2 = new Employee() { id = 1, name = "ali", age = 20 };

em1.Equals(em2);  // Returns FALSE - different objects in memory
```

Even though the values are identical, they're different objects in memory.

Custom Value Equality

Your `Employee` class overrides `Equals()` to compare actual values:

```
public override bool Equals(object? obj)
{
    Employee em = (Employee)obj;
    return (id == em.id && name == em.name && age == em.age);
}
```

Now the comparison checks if the **content** is the same:

```
Employee em1 = new Employee() { id = 1, name = "ali", age = 20 };
Employee em2 = new Employee() { id = 1, name = "ali", age = 20 };

em1.Equals(em2);  // Returns TRUE - same values
```

Structs and Equals

For structs (like `Complex`), the default `Equals()` behavior is different:

Default Behavior: Structs inherit from `ValueType`, which overrides `Equals()` to compare all fields using reflection.

Performance: The default struct comparison uses reflection, which is slow.

Recommendation: Override `Equals()` in structs for better performance.

```
Complex c1 = new Complex() { real = 1, img = 2 };
Complex c2 = new Complex() { real = 1, img = 2 };

c1.Equals(c2);  // Returns TRUE (default struct behavior compares fields)
```

Proper Equals Implementation

A robust `Equals()` override should:

1. Check for null:

```
if (obj == null) return false;
```

2. Check for reference equality (optimization):

```
if (ReferenceEquals(this, obj)) return true;
```

3. Check for type compatibility:

```
if (obj.GetType() != this.GetType()) return false;
```

4. Cast and compare:

```
Employee em = (Employee)obj;
return id == em.id && name == em.name && age == em.age;
```

Better implementation:

```
public override bool Equals(object? obj)
{
    if (obj == null || GetType() != obj.GetType())
        return false;

    if (ReferenceEquals(this, obj))
        return true;
```

```
    Employee em = (Employee)obj;
    return id == em.id && name == em.name && age == em.age;
}
```

Equals Contract Rules

When overriding `Equals()`, you must maintain these rules:

Reflexive: `x.Equals(x)` must return true.

Symmetric: If `x.Equals(y)` is true, then `y.Equals(x)` must be true.

Transitive: If `x.Equals(y)` and `y.Equals(z)` are true, then `x.Equals(z)` must be true.

Consistent: Multiple calls to `x.Equals(y)` should return the same value.

Null Handling: `x.Equals(null)` must return false.

Important Warning

Always override GetHashCode when overriding Equals. Objects that are equal must have the same hash code. If you override `Equals()` but not `GetHashCode()`, collections like `Dictionary` and `HashSet` won't work correctly.

GetHashCode Method Override

What is a Hash Code?

A hash code is an integer value that represents the object's content. It's used by hash-based collections (Dictionary, HashSet, Hashtable) to organize and quickly locate objects.

Purpose of GetHashCode

Fast Lookups: Hash codes enable O(1) average time complexity for lookups in hash tables.

Bucketing: Objects are placed into "buckets" based on their hash code.

Equality Optimization: Before checking full equality, collections can quickly compare hash codes.

Default Behavior

For reference types, the default `GetHashCode()` returns a unique value based on the object's memory location. For value types, it's based on the fields.

Your Implementation

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

This implementation uses only the `id` field as the hash code. This is simple but may not be optimal if `id` distribution is poor.

Hash Code Contract Rules

Equal Objects Must Have Equal Hash Codes: If `a.Equals(b)` is true, then `a.GetHashCode()` must equal `b.GetHashCode()`.

Reverse Not Required: Objects with the same hash code don't have to be equal (collisions are allowed).

Consistency: Hash code should remain constant during object's lifetime.

Performance: Should be fast to compute.

Distribution: Should distribute values evenly to minimize collisions.

Why Are Hash Codes Important?

In your code example:

```
Employee em = new Employee() { id = 2, name = "ali" };
Employee em1 = new Employee() { id = 4, name = "ali" };
em1 = em;  // em1 now references the same object as em

Console.WriteLine(em.GetHashCode());   // Output: 2
Console.WriteLine(em1.GetHashCode());  // Output: 2 (same object)
```

Both return `2` because they reference the **same object** and your hash code is based on the `id` field.

Better GetHashCode Implementation

For multiple fields, you should combine them:

Simple Approach:

```
public override int GetHashCode()
{
    return id.GetHashCode() ^ name.GetHashCode() ^ age.GetHashCode();
}
```

Better Approach (C# 9+):

```
public override int GetHashCode()
{
    return HashCode.Combine(id, name, age);
}
```

Traditional Approach:

```
public override int GetHashCode()
{
    unchecked
    {
        int hash = 17;
        hash = hash * 23 + id.GetHashCode();
        hash = hash * 23 + (name?.GetHashCode() ?? 0);
        hash = hash * 23 + age.GetHashCode();
        return hash;
    }
}
```

Hash Collisions

A **collision** occurs when different objects produce the same hash code. This is normal and unavoidable (pigeonhole principle: infinite objects, finite hash codes).

Collections handle collisions by:

- Storing multiple objects in the same bucket
- Using `Equals()` to determine actual equality when hash codes match

Common Mistakes

Mutable Hash Codes: Don't base hash codes on mutable fields if the object is used as a dictionary key.

```
Employee em = new Employee() { id = 1, name = "ali" };
dictionary.Add(em, "data");
em.id = 2;  // ❌ Now the dictionary can't find this object!
```

Not Overriding with Equals: If you override `Equals()` but not `GetHashCode()`, hash-based collections will break.

Returning Constant: Never do `return 0;` - this creates a hash table with one giant bucket, destroying performance.

GetType Method

What is GetType?

`GetType()` is a method that returns the runtime type of the current instance. It returns a `Type` object that contains metadata about the type.

Key Characteristics

Sealed Method: Cannot be overridden (marked as sealed in Object class).

Runtime Type: Returns the actual type of the object, not the declared type.

Reflection Entry Point: The `Type` object is the starting point for reflection operations.

Usage in Your Code

```
Employee emp = new Employee();
Console.WriteLine(emp.GetType().BaseType);
// Output: System.Object

Complex c = new Complex();
Console.WriteLine(c.GetType().BaseType.BaseType);
// Output: System.Object
// BaseType chain: Complex -> ValueType -> Object
```

Type Hierarchy

For Classes:

```
Employee -> Object
```

For Structs:

```
Complex -> ValueType -> Object
```

All value types inherit from `System.ValueType`, which inherits from `System.Object`.

Type Object Properties and Methods

The `Type` object provides extensive information:

Basic Information:

```
Type t = emp.GetType();
Console.WriteLine(t.Name);          // "Employee"
Console.WriteLine(t.FullName);      // "Day4PII.Employee"
Console.WriteLine(t.Namespace);     // "Day4PII"
Console.WriteLine(t.Assembly);      // Assembly information
Console.WriteLine(t.IsClass);       // True
Console.WriteLine(t.IsValueType);   // False
Console.WriteLine(t.IsSealed);      // False
Console.WriteLine(t.IsAbstract);    // False
```

Hierarchy Information:

```
Console.WriteLine(t.BaseType);             // System.Object
Console.WriteLine(t.GetInterfaces());      // Implemented interfaces
```

Members:

```
Console.WriteLine(t.GetProperties());      // All properties
Console.WriteLine(t.GetMethods());         // All methods
Console.WriteLine(t.GetFields());          // All fields
Console.WriteLine(t.GetConstructors());    // All constructors
```

GetType vs typeof

GetType(): Runtime type of an instance

```
Employee em = new Employee();
Type t1 = em.GetType();  // Requires an instance
```

typeof(): Compile-time type information

```
Type t2 = typeof(Employee);  // No instance needed
```

Both return the same `Type` object for the same type.

Polymorphism and GetType

```
object obj = new Employee();
Console.WriteLine(obj.GetType());  // Output: Day4PII.Employee (actual type)
```

Even though `obj` is declared as `object`, `GetType()` returns `Employee` because that's the actual runtime type.

Practical Uses

Type Checking:

```
if (obj.GetType() == typeof(Employee))
{
    Employee em = (Employee)obj;
}
```

Dynamic Type Discovery:

```
Type t = obj.GetType();
MethodInfo method = t.GetMethod("ToString");
object result = method.Invoke(obj, null);
```

Reflection Operations:

```
Type t = typeof(Employee);
object instance = Activator.CreateInstance(t);  // Create instance dynamically
```

Comparing Types:

```
if (obj1.GetType() == obj2.GetType())
{
    // Same type
}
```

GetType vs is/as Operators

GetType(): Exact type match only

```
object obj = new Employee();
obj.GetType() == typeof(Employee)  // True
obj.GetType() == typeof(object)    // False
```

is operator: Checks if compatible (includes inheritance)

```
obj is Employee   // True
obj is object     // True (Employee inherits from object)
```

as operator: Safe casting

```
Employee em = obj as Employee;  // Returns null if cast fails
```

Boxing and Unboxing

What is Boxing?

Boxing is the process of converting a value type to a reference type (specifically to `object` or an interface type). The value is wrapped in an object and allocated on the heap.

Boxing Example

```
int i = 123;            // Value type on stack
object obj = i;         // Boxing: int -> object (now on heap)
```

What happens during boxing:

1. Memory is allocated on the managed heap
2. The value is copied from the stack to the heap
3. A reference to the heap location is returned
4. The original stack value remains unchanged

Boxing in Your Code

```
object obj = new Employee();  // No boxing (already reference type)
obj = 1;                      // Boxing! int -> object
```

```
obj = new Complex();          // Boxing! Complex struct -> object
obj = "ali";                  // No boxing (string is reference type)
```

```
object[] arr = new object[4];
arr[0] = 1;              // Boxing: int -> object
arr[1] = new Employee(); // No boxing
```

What is Unboxing?

Unboxing is the explicit conversion from an object back to a value type. The value is extracted from the heap object and copied to the stack.

Unboxing Example

```
object obj = 123;       // Boxing
int i = (int)obj;       // Unboxing: object -> int
```

Unboxing Rules

Must be explicit: Requires a cast.

Must be correct type: Cannot unbox to a different type.

```
object obj = 123;
long l = (long)obj;  // ❌ Runtime error! Must unbox to int first
int i = (int)obj;
long l = i;          // ✓ Now conversion is fine
```

Null reference: Unboxing null throws `NullReferenceException`.

```
object obj = null;
int i = (int)obj;  // ❌ NullReferenceException
```

Performance Impact of Boxing

Heap Allocation: Boxing requires memory allocation on the heap (expensive).

Garbage Collection: Boxed values create objects that need to be garbage collected.

Copying Overhead: Values are copied during boxing and unboxing.

Performance Hit: Can be 10-20x slower than working with value types directly.

Example of Performance Problem

```
// Poor performance - multiple boxing operations
ArrayList list = new ArrayList();
for (int i = 0; i < 1000; i++)
{
    list.Add(i);  // Boxing on each iteration!
}

// Better - using generics (no boxing)
List<int> list = new List<int>();
for (int i = 0; i < 1000; i++)
{
    list.Add(i);  // No boxing!
}
```

Avoiding Boxing

Use Generics: Generic collections and methods avoid boxing.

```
List<int> instead of ArrayList
Dictionary<int, string> instead of Hashtable
```

Use Value Types Directly: Don't store value types in object variables unless necessary.

Avoid Object Parameters: Use generic parameters instead.

```
// Boxing
void Process(object obj) { }
Process(123);

// No boxing
void Process<T>(T value) { }
Process(123);
```

Detecting Boxing

Look for these patterns:

- Value types assigned to `object` variables
- Value types passed to methods that accept `object`
- Value types stored in non-generic collections
- Value types in string concatenation: `"Value: " + 123` (boxes 123)

- Value types implementing interfaces and stored as interface type

Boxing with Interfaces

When a struct implements an interface and is cast to that interface, boxing occurs:

```
interface IProcess { void Run(); }

struct MyStruct : IProcess
{
    public void Run() { }
}

MyStruct s = new MyStruct();
IProcess p = s;  // Boxing! Struct -> Interface reference
```

Reference Types vs Value Types

Fundamental Differences Summary

Aspect	Value Types (struct)	Reference Types (class)
Storage	Stack (usually)	Heap
Contains	Actual data	Reference (pointer) to data
Assignment	Copies entire value	Copies reference
Default value	All fields initialized to default	null
Inheritance	Cannot inherit from other types	Can inherit from classes
Null	Cannot be null (unless Nullable)	Can be null
Performance	Faster allocation/deallocation	More overhead
Memory	More efficient for small data	Better for large, complex data

Memory Layout Example

```
// Value type
int x = 5;
int y = x;     // Copies the value
y = 10;        // x remains 5

// Reference type
```

```
Employee em1 = new Employee() { id = 1 };
Employee em2 = em1;  // Copies the reference
em2.id = 2;          // em1.id is also 2 (same object)
```

Stack vs Heap

Stack:

- LIFO (Last In, First Out) structure
- Fast allocation and deallocation
- Limited size
- Automatically cleaned up when scope exits
- Thread-specific

Heap:

- Managed memory area
- Slower allocation
- Much larger size
- Cleaned up by garbage collector
- Shared across threads

Reference Assignment Behavior

Your code demonstrates this:

```
Employee em = new Employee() { id = 2, name = "ali" };
Employee em1 = new Employee() { id = 4, name = "ali" };
em1 = em;  // em1 now points to the same object as em

Console.WriteLine(em.GetHashCode());   // 2
Console.WriteLine(em1.GetHashCode());  // 2 (same object!)
```

After `em1 = em`:

- The original object with id=4 becomes eligible for garbage collection
- Both `em` and `em1` reference the object with id=2
- Changes through either variable affect the same object
- GetHashCode returns the same value because it's the same object

Value Type Assignment Behavior

```
Complex c1 = new Complex() { real = 1, img = 2 };
Complex c2 = c1;  // Creates independent copy
c2.real = 5;      // Only c2 is modified
// c1.real is still 1
```

Null and Value Types

Value types cannot be null by default:

```
int x = null;      // ❌ Compile error
Employee em = null; // ✓ OK (reference type)
```

To allow null for value types, use `Nullable<T>` or `T?`:

```
int? x = null;           // ✓ OK
Nullable<int> y = null;  // ✓ Same thing
```

Passing Parameters

Value Types (by default pass by value):

```
void Modify(int x)
{
    x = 10;  // Only modifies local copy
}

int num = 5;
Modify(num);
// num is still 5
```

Reference Types (pass reference by value):

```
void Modify(Employee em)
{
    em.id = 10;  // Modifies the original object
}

Employee emp = new Employee() { id = 5 };
Modify(emp);
// emp.id is now 10
```

Ref Keyword (pass by reference):

```csharp
void Modify(ref int x)
{
    x = 10;  // Modifies original variable
}

int num = 5;
Modify(ref num);
// num is now 10
```

When Each Type is Appropriate

Use Value Types (struct) for:

- Mathematical values (coordinates, vectors, complex numbers)
- Small data structures (< 16 bytes)
- Types that represent a single value
- When you need independent copies
- When performance is critical for small, frequently used types

Use Reference Types (class) for:

- Entities with identity (employees, customers, orders)
- Large data structures
- When you need polymorphism
- When multiple variables should reference the same data
- When the object has a long lifetime

Common Value Types in .NET

- All numeric types: `int`, `long`, `float`, `double`, `decimal`
- `bool`, `char`
- `DateTime`, `TimeSpan`
- `Guid`
- All enums
- Custom structs

Common Reference Types in .NET

- `string` (special case: immutable reference type)
- `object`
- Arrays

- Delegates
- All classes
- Interfaces (used as reference types)

Advanced Concepts

Immutability and Value Types

Best practice for structs is to make them immutable:

```
struct ImmutableComplex
{
    public int Real { get; }
    public int Img { get; }

    public ImmutableComplex(int real, int img)
    {
        Real = real;
        Img = img;
    }
}
```

Benefits:

- Thread-safe by default
- No unexpected mutations
- Can be safely shared
- Works better with hash-based collections

The `readonly` Modifier for Structs

C# 7.2+ introduced `readonly struct`:

```
readonly struct ReadOnlyComplex
{
    public int Real { get; }
    public int Img { get; }

    public ReadOnlyComplex(int real, int img)
    {
        Real = real;
        Img = img;
```

```
    }
}
```

The compiler enforces immutability and can optimize more aggressively.

Records (C# 9+)

Records provide a concise way to create immutable reference types:

```
record Employee(int Id, string Name, int Age);

// Automatically implements:
// - Value-based equality
// - ToString
// - GetHashCode
// - Copy constructor
```

The Dangers of Mutable Structs

Mutable structs can cause confusing behavior:

```
struct MutablePoint
{
    public int X { get; set; }
    public int Y { get; set; }
}

class Container
{
    public MutablePoint Point { get; set; }
}

Container c = new Container();
c.Point.X = 5;  // ❌ Compiles but doesn't work as expected!
// You're modifying a temporary copy returned by the property
```

Equality Operators

In addition to `Equals()`, you can override equality operators:

```
public static bool operator ==(Employee left, Employee right)
{
    if (ReferenceEquals(left, right)) return true;
    if (left is null || right is null) return false;
```

```
    return left.Equals(right);
}

public static bool operator !=(Employee left, Employee right)
{
    return !(left == right);
}
```

Now you can use:

```
if (em1 == em2)  // Uses operator
if (em1.Equals(em2))  // Uses method
```

By Abdullah Ali

Contact : +201012613453