.NET

System.Object Class

Introduction

The `Object` class (System.Object) serves as the ultimate base class for all types in the .NET type system, introduced in the very first version of the framework (2002). As the root of the type hierarchy, it provides common functionality that every .NET type inherits, ensuring a consistent behavioral contract across all objects.

Core Functionality and Members

1. Common Methods

- All objects inherit these fundamental methods:
  - `Equals()`: Supports value equality comparison
  - `GetHashCode()`: Provides hash code generation
  - `ToString()`: Offers string representation
  - `GetType()`: Returns runtime type information

2. Virtual Methods

- Key overridable methods that enable polymorphism:
  - `Equals()`: Default implementation uses reference equality
  - `GetHashCode()`: Should be overridden when Equals is overridden
  - `ToString()`: Default returns fully qualified type name
  - `Finalize()`: Cleanup method - Considered deprecated and should be avoided in favor of IDisposable

3. Final Methods

- Non-overridable critical operations:
  - `GetType()`: Prevents tampering with type identity

    - `MemberwiseClone()`: Provides shallow copying capability

Type System Integration

1. Unified Type Hierarchy

- Value types inherit via System.ValueType (which inherits from Object)
- Reference types inherit directly or indirectly from Object
- Enables polymorphic treatment of all types

2. Boxing Mechanism

- Facilitates value type to reference type conversion:
    - Value types get boxed when cast to Object
    - Creates heap allocation and copy overhead

3. Type Safety Foundation

- Provides the basis for:
    - Runtime type checking
    - Reflection capabilities
    - Safe casting operations

Common Usage Patterns

1. Generic Programming

Serves as constraint-less upper bound:

```
void Process(object item) { ... }  // Accepts any type
```

2. Collections (Pre-Generics)

Non-generic collections (ArrayList, Hashtable) used Object as element type:

```
ArrayList list = new ArrayList();
list.Add(42);  // Boxing occurs
```

3. Reflection Scenarios

Enables type-agnostic processing:

```
object instance = Activator.CreateInstance(someType);
```

Best Practices

1. Method Overriding

- When overriding Equals():
    - Maintain reflexivity, symmetry, and transitivity
    - Override GetHashCode() consistently
    - Consider implementing IEquatable

2. ToString() Implementation

- Provide meaningful string representation
- Include all significant state information
- Keep culture-invariant for machine consumption

3. Type Checking

- Prefer pattern matching over direct GetType() checks:

```
if (obj is string s) { ... }  // Modern approach
```

Performance Considerations

1. Boxing Overhead

- Value type to Object conversion is expensive
- Generics (List) eliminate this overhead

2. Virtual Call Impact

- Virtual method calls have slight overhead
- Sealed classes can optimize this

3. Hash Code Generation

- Poor GetHashCode() implementations hurt hash tables
- Should be fast and produce well-distributed values

MSDN References

- Object Class (System)
- Object Lifetime
- Type System Fundamentals

Interfaces

1. What is an Interface?

1.1 Definition

- A **contract** that defines a set of methods/properties a class **must** implement.
- Pure abstraction (no implementation until C# 8.0).

1.2 Key Characteristics

- **No fields** (only methods, properties, events, indexers).
- **No constructors** (cannot be instantiated directly).
- **Multiple inheritance** (a class can implement many interfaces).

1.3 Syntax

```
public interface ILogger
{
    void Log(string message);
    string LogLevel { get; set; }  // Property
}
```

2. Why Use Interfaces?

2.1 Design Benefits

- **Decoupling**: Code depends on abstractions, not concrete classes.
- **Polymorphism**: Different classes can be treated uniformly.
- **Testability**: Easy mocking for unit tests.

2.2 Real-World Analogy

- **Interface** → USB port (standardized contract).
- **Implementing Class** → Device (e.g., phone, laptop).

3. Implementing Interfaces

3.1 Basic Implementation

```
public class FileLogger : ILogger {
    public string LogLevel { get; set; }
    public void Log(string message) {
        File.WriteAllText("log.txt", message);
    }
}
```

3.2 Explicit Implementation

- Avoids naming conflicts when implementing multiple interfaces.

```
public class HybridLogger : ILogger, IDisposable
{
    void ILogger.Log(string message) { /* ILogger's Log */ }
    void IDisposable.Dispose() { /* IDisposable's Dispose */ }
}

// Usage:
ILogger logger = new HybridLogger();
logger.Log("Test");  // Calls ILogger.Log
```

4. Interface Inheritance

4.1 Chaining Interfaces

```
public interface IAuditable
{
    void Audit();
}

public interface IAdvancedLogger : ILogger, IAuditable
{
    void LogError(Exception ex);
}
```

4.2 Rules

- **Interfaces can inherit other interfaces**.
- **Classes must implement all parent interface members**.

5. Interfaces vs. Abstract Classes

Feature	Interface	Abstract Class
Inheritance	Multiple	Single
Implementation	None (until C# 8.0)	Partial
Fields	✗	✓
Constructors	✗	✓

6. Common .NET Interfaces

Interface	Purpose	Key Method
`IEnumerable`	Iteration	`GetEnumerator()`
`IDisposable`	Resource cleanup	`Dispose()`

Interface	Purpose	Key Method
`IComparable`	Sorting	`CompareTo()`
`IComparer`	Sorting	`Compare()`

7. Design Patterns with Interfaces

7.1 Strategy Pattern

```
public interface IPaymentStrategy {
    void ProcessPayment(double amount);
}

public class CreditCardPayment : IPaymentStrategy { ... }
public class PayPalPayment : IPaymentStrategy { ... }

// Usage:
IPaymentStrategy strategy = new CreditCardPayment();
strategy.ProcessPayment(100);
```

7.2 Dependency Injection

```
public class OrderService {
    private readonly ILogger _logger;
    // Constructor injection
    public OrderService(ILogger logger) {
        _logger = logger;
    }
}
```

8. Best Practices

- **Prefix with `I`** (e.g., `ILogger`).
- **Keep interfaces small** (Single Responsibility Principle).
- **Favor interfaces for cross-cutting concerns** (logging, caching).

9. MSDN References

- Interfaces (C#)
- Explicit Interface Implementation

Generics in C# .NET

Introduction

Generics were introduced in C# 2.0 (.NET Framework 2.0, 2005) as a revolutionary feature to address several limitations of the original type system. This implementation was influenced by templates in C++ but designed with .NET's runtime characteristics in mind.

Core Concepts and Definitions

1. Generic Type Parameters

Generics allow the definition of type parameters (denoted by `<T>`) that serve as placeholders for actual types. These parameters enable:

- Creation of classes, interfaces, methods and delegates that work with any data type
- Compile-time type safety without sacrificing performance
- Elimination of runtime type checking and casting

2. Better Generics Implementation

- Unlike Java's type erasure, .NET implements generics at:
  - Runtime level (CLR understands generics)

- JIT compilation level (specialized native code generation)
- Metadata level (preserved in assemblies)

3. Performance Benefits

- Generics provide significant advantages over non-generic approaches:
  - Eliminate boxing for value types
  - Reduce runtime type checks
  - Enable more efficient code generation

Generic Type Definitions

1. Generic Classes

Classes can be defined with one or more type parameters:

```
public class Box<T> {
    private T val;
    public void Set(T value) { this.val = value; }
    public T Get() { return this.val; }
    public void Display() {
        Console.WriteLine(val);
    }
}
```

```
Box<string> b1 = new Box<string>(); // generic type given = string
b1.Set("Secret");
string str = b1.Get();
// ...

Box<int> b2 = new Box<int>(); // generic type given = string
```

```
b2.Set(123);
int num = b2.Get();
// ...
```

2. Generic Methods

- Methods can have their own type parameters independent of the containing class.
- Non-generic class may have generic methods:

```
class Util {
    public void Swap<T>(ref T x, ref T y) {
        T t = x;
        x = y;
        y = t;
    }
}

// Usage
Util util = new Util();
double n1 = 1.1, n2 = 2.2;
util.Swap(ref n1, ref n2); // generic type is inferred = double
Console.WriteLine($"n1: {n1}, n2: {n2}");
```

- Generic class may have generic methods e.g. Set(), Get() in Box class.
- Generic class may have non-generic methods e.g. Display() in Box class.

3. Generic Interfaces

- Interfaces can define type parameters for implementing classes:

```
public interface IRepository<T> {
    IEnumerable<T> FindAll();
    void Add(T entity);
}
```

- Built-in examples: `IComparable<T>`, `IComparer<T>`, etc.
- Generic class/interface can be inherited into non-generic class.

```
class Emp : IComparable<Emp>
{
    public int Id { get; set; }
    public String Name { get; set; }
    public double Salary { get; set; }

    public int CompareTo(Emp? other) {
        if (other == null)
            return 1;
        int diff = this.Id - other.Id;
        return diff;
    }

    public override string ToString() {
        return $"Emp: Id={Id}, Name={Name}, Salary={Salary}";
    }
}
```

```
// Usage
Emp[] arr = new Emp[5]
{
    new Emp { Id = 2, Name = "John", Salary = 2000.0 },
    new Emp { Id = 5, Name = "Mark", Salary = 1500.0 },
    new Emp { Id = 1, Name = "Steve", Salary = 3500.0 },
    new Emp { Id = 4, Name = "Peter", Salary = 4000.0 },
    new Emp { Id = 3, Name = "Tony", Salary = 3000.0 }
};

Console.WriteLine("Emps Sorted by Id: ");
Array.Sort(arr);
foreach (Emp x in arr)
    Console.WriteLine(x.ToString());
```

4. Generic Delegates

- We will discuss this later.

Constraints

1. Constraint Types

- Constraints restrict what types can be used as arguments:
    - `where T : struct` (value type)
    - `where T : class` (reference type)
    - `where T : new()` (default constructor)
    - `where T : BaseClass` (specific base class)
    - `where T : ISomeInterface` (interface implementation)

2. Examples

```
public static T Max<T>(T first, T second) where T : IComparable<T> {
    return first.CompareTo(second) > 0 ? first : second;
}
```

```
// Usage : If Emp is implements IComparable<Emp>
Emp e1 = new Emp { Id = 1, Name = "Steve", Salary = 3500.0 };
Emp e2 = new Emp { Id = 4, Name = "Peter", Salary = 4000.0 };
Emp me = Max(e1, e2);
```

3. Multiple Constraints

- A type parameter can have multiple constraints:

  ```
  public T CreateInstance<T>() where T : class, new()
  {
      return new T();
  }
  ```

Limitations

- Generic type parameters cannot be used for:
  - Static fields/methods specific to the constructed type
  - Operator overloading
  - Certain reflection operations

Advanced Generic Patterns

1. Covariance/Contravariance

- Introduced in C# 4.0 for interfaces/delegates:
  - Covariance (`out T`) allows more derived types
  - Contravariance (`in T`) allows less derived types

2. Generic Variance

- Enables more flexible type relationships:

```
IEnumerable<string> strings = new List<string>();
IEnumerable<object> objects = strings;  // Covariant
```

3. Default Values

- The `default` keyword handles null/non-null cases:

```
T value = default(T);  // null for classes, zero for structs
```

Performance Considerations

1. Code Generation

- The JIT compiler generates:
  - Separate code for each (used) value type
  - Shared code for reference types

2. Memory Efficiency

- Generics eliminate:
  - Boxing overhead for value types
  - Redundant code for similar operations

3. Runtime Efficiency

- Provides:
  - Direct method calls without casting

  - Optimized collections for value types

Best Practices

1. Naming Conventions

- Single type parameters: T, TResult
- Multiple parameters: TSource, TResult
- Descriptive when needed: TEntity, TRepository

2. Constraint Usage

- Apply constraints:
  - As loosely as possible
  - Only when necessary

3. Design Considerations

- Prefer generic methods over whole generic classes when possible
- Consider generic base classes for shared functionality
- Document type parameter requirements

MSDN References

- Generics (C# Programming Guide)
- Generic Classes and Methods
- Constraints on Type Parameters

Delegates in C#

1. What is a Delegate?

1.1 Definition

- A **delegate** is a **type-safe function pointer** that references methods with a specific signature.
- Enables **callback mechanisms**, **event handling**, and **dynamic method invocation**.

1.2 Key Properties

- **Type-safe**: Compiler enforces method signatures.
- **Can reference static or instance methods**.
- **Multicast capable** (invokes multiple methods sequentially).

2. Delegate Declaration & Usage

2.1 Basic Syntax

```
// Step 1: Declare a delegate type
public delegate void LogMessage(string message);

// Step 2: Create a delegate instance (built-in method)
LogMessage logger = new LogMessage(Console.WriteLine);

// Step 3: Invoke
logger("Hello, delegates!");  // Calls Console.WriteLine
```

2.2 Assigning Methods

```
// Static method (user-defined method)
static void LogToFile(string msg) {
    File.WriteAllText("log.txt", msg);
}
```

```
// Step 2: Initialize delegate instance (to user-defined static method)
LogMessage fileLogger = new LogMessage(LogToFile);

// Step 3: Invoke
fileLogger("Hello, delegates!");  // Calls LogToFile()
```

```
// Define Instance method (user-defined method)
class NotificationService {
    // fields
    public void SendEmail(String msg) {
        // Send email
    }
}

// Step 2: Initialize delegate instance (to user-defined non-static method)
NotificationService service = new NotificationService();
LogMessage emailLogger = new LogMessage(service.SendEmail);
//OR
LogMessage emailLogger = service.SendEmail; // Implicit syntax - Internally allocates delegate object

// Step 3: Invoke
emailLogger("Hello, delegates!");  // Calls service.SendEmail()
```

3. Multicast Delegates

3.1 Chaining Methods

- Use `+=` to add and `-=` to remove methods.

```
LogMessage multiLogger = Console.WriteLine;
multiLogger += LogToFile;
multiLogger += service.SendEmail;

multiLogger("This goes to both!");  // Calls all three methods
```

3.2 Return Values in Multicast

- Only the **last method's return value** is captured.

```
public delegate int MathOp(int x, int y);

public int Add(int a, int b) {
    return a + b;
}

MathOp operations = Add;
operations += Subtract;
operations += Multiply;

int result = operations(3, 4);  // Returns 12 (Multiply's result)
```

- To access return values of individual delegates use GetInvocationList().

```
Delegate[] delegates = operations.GetInvocationList();
foreach (MathOp del in delegates) {
    int result = del(22, 7);
    Console.WriteLine(result);
}
```

4. Delegate Use Cases

4.1 Callbacks

```
delegate void StatusHandler(string status);

public void ProcessData(StatusHandler callback) {
    // ...
    callback("Success!");
}

// Usage
ProcessData(Console.WriteLine);
```

4.2 Strategy Pattern

```
public delegate int PaymentStrategy(double amount);

public class PaymentProcessor {
    public void Process(PaymentStrategy strategy, double amount) {
        int status = strategy(amount);
        Console.WriteLine($"Status: {status}");
    }
}

class Program {
    static void Main(string[] args) {
        string PayByCreditCard(double amount) {
            // ...
            return "Success";
        }

        string PayByUPI(double amount) {
            // ...
            return "Failed";
        }

        PaymentProcessor processor = new PaymentProcessor();
        processor.Process(PayByCreditCard, 2000.0);
        processor.Process(PayByUPI, 4000.0);
    }
}
```

5. Best Practices

- **Use `Action`/`Func`** for common cases (avoid custom delegates) - (we'll cover later).
- **Always check for `null` before invocation**:

  ```
  logger?.Invoke("Safe call");
  ```

- **Prefer events for pub-sub patterns** (we'll cover later).

6. MSDN References

- Delegates (C#)

Func/Action/Predicate Delegates

1. Evolution

1.1 Traditional Delegates

- Since C# 1.0 (2002)
- **Type-safe function pointers** that reference methods with specific signatures.

- Must be declared before use i.e. Required explicit delegate type declarations.
- Example:

```
delegate int MathOperation(int a, int b);  // Custom delegate type
int Add(int a, int b) {
  return a + b;
}
MathOperation add = new MathOperation(Add);
```

1.2 Modern Generic Delegates

- In C# 3.0 (2007)
- Introduced `Func`, `Action`, and `Predicate` to reduce boilerplate.
- Part of the **System** namespace.

2. Generic Delegates (`Func`, `Action`, `Predicate`)

Delegate	Purpose	Signature
`Action`	For void methods	`Action<T1, T2>`
`Func`	For methods with return values	`Func<T1, T2, TResult>`
`Predicate`	For boolean conditions	`Predicate<T>`

3.1 `Action` Delegate

- **Purpose**: For methods that **return void**.
- **Overloads**: `Action`, `Action<T>`, `Action<T1,T2>`, ..., up to 16 parameters.
- **Example**:

```
Action<string> log = Console.WriteLine;
log("Hello, Action!");
```

3.2 `Predicate` Delegate

- **Purpose**: For methods that **return a boolean** i.e. test a condution.
- Legacy delgate: largely replaced by `Func<T, bool>`.
- **Example**:

```
boolean IsEven(int x) {
  return x % 2 == 0;
}
Predicate<int> condition = IsEven;
bool flag = condition(4);  // true
```

3.3 `Func` Delegate

- **Purpose**: For methods that **return a value**.
- **Last type parameter**: Return type.
- **Example**:

```
int Add(int x, int y) {
  return x + y;
}
Func<int, int, int> add = Add;
int result = add(5, 7);  // 12
```

Comparison with Java's Functional interfaces

C# Delegate	Java Functional Interface	Description
`Action<T>`	`Consumer<T>`	Method with no return value (void)
`Func<TResult>`	`Supplier<TResult>`	Method with no arguments, returns a value
`Func<T, TResult>`	`Function<T, TResult>`	Method with one argument, returns a value
`Func<T1, T2, TResult>`	`BiFunction<T1, T2, TResult>`	Method with two arguments, returns a value
`Predicate<T>`	`Predicate<T>`	Method with one argument, returns boolean

3. When to Use Which

3.1 Prefer `Func`/`Action` When:

- Needing **quick, inline & standard delegate definitions**.
- Working with **lambda expressions**.
- Using **LINQ** or functional programming patterns.

3.2 Prefer Traditional Delegates When:

- Defining **event handlers** (e.g., `public delegate void EventHandler()`).
- Requiring **explicit naming** for API clarity.
- Handling **specialized signatures** (e.g., `ref`/`out` parameters).
- **Specialized signatures** not covered by `Func`/`Action`.

4. Performance Considerations

- **No runtime performance difference**: All delegates compile to similar IL.
- **Memory**: Traditional delegates may marginally increase metadata size.

5. Best Practices

✔ **Use `Func`/`Action`** for lambda expressions and LINQ.
✔ **Reserve traditional delegates** for events and public APIs.
✔ **Replace `Predicate<T>`** with `Func<T, bool>` for consistency.

6. MSDN References

- Func Delegate
- Action Delegate

Anonymous Methods and Lambda Expressions

1. Introduction

1.1 Introduction Timeline

- **C# 2.0 (2005)**: Introduced **anonymous methods** for simplified delegate syntax.
- **C# 3.0 (2007)**: Added **lambda expressions**, superseding anonymous methods in most cases.

1.2 Purpose

- Reduce boilerplate code for delegate instantiation.
- Enable functional programming patterns (e.g., closures, higher-order functions).

2. Anonymous Methods

2.1 Definition

- Inline delegate definitions without a named method.
- Syntax: `delegate(parameters) { ... }`

2.2 Key Characteristics

✔ **No return type declaration** (inferred from context).
✔ **Can omit parameters** if unused (`delegate { ... }`).
✔ **Require explicit `return`** for non-void methods.

2.3 Example

```
// Traditional delegate (C# 1.0)
delegate void Printer(string s);
Printer pr1 = delegate(string msg) {
        Console.WriteLine(msg);
    };
pr1("Hi");

// Parameterless delegate (System.Action type)
Action pr2 = delegate {
        Console.WriteLine("Hello");
    };
pr2();
```

2.4 Limitations

✖ Cannot use `yield return`.
✖ No support for expression-bodied syntax.

---

3. Lambda Expressions

3.1 Definition

- Concise syntax for anonymous methods: `(parameters) => expression-or-block`.
- Two forms:
  - **Expression lambdas**: Single-line

```
x => x * x
```

- **Statement lambdas**: Multi-line

```
x => {
    int res = x * x;
    return res;
}
```

3.2 Type Inference

- Compiler infers types from context (e.g., `Func<int, int>` for `x => x + 1`).
- Explicit types can be specified: `(int x) => x + 1`.

3.3 Examples

Scenario	Lambda Syntax
Square a number	`x => x * x`
Filter even numbers	`n => n % 2 == 0`
Multi-line processing	`s => { s = s.Trim(); return s; }`

3.4 Common Delegate Types

Delegate	Lambda Example	Purpose
`Action`	`() => Console.WriteLine()`	Void methods
`Func<T>`	`() => 42`	Return value

Delegate	Lambda Example	Purpose
`Predicate<T>`	`x => x > 0`	Boolean conditions

---

4. Closures and Captured Variables

4.1 Definition

- **Closure**: A lambda/anonymous method that captures variables from its enclosing scope.
- **Lifetime**: Captured variables persist until the delegate instance is garbage-collected.

4.2 Example

```
public Func<int> CreateCounter()
{
    int count = 0;
    return () => ++count;  // Captures 'count'
}

// Usage:
var counter = CreateCounter();
Console.WriteLine(counter());  // 1
Console.WriteLine(counter());  // 2
```

4.3 Pitfalls

✔ **Avoid modifying captured variables** in multi-threaded contexts.
✔ **Memory usage**: Long-lived delegates keep captured variables alive.

---

5. Lambda vs. Anonymous Method Comparison

Feature	Lambda Expressions	Anonymous Methods
Syntax	`x => x + 1`	`delegate(int x) { return x + 1; }`
Type Inference	Full support	Limited
Expression-bodied	Yes	No
LINQ Compatibility	Preferred	Rarely used

---

6. Best Practices

✔ **Prefer lambdas** over anonymous methods (modern syntax).
✔ **Avoid complex logic** in lambdas (extract to methods if 3+ lines).

7. MSDN References

- Anonymous Methods
- Lambda Expressions (C#)
- Closures
