The `object` Type

In C#, the `object` type (which is an alias for `System.Object`) is the **ultimate base class for all other types**. This means that every type you create or use in C#, whether it's a built-in type like `int` or `string`, a custom class, or even a value type like a `struct`, implicitly or explicitly derives from `object`.

Why is this useful?

This universal base class allows for great flexibility, particularly in scenarios where you need to work with data of unknown or mixed types. Consider a general-purpose `Stack` data structure, which operates on the principle of LIFO (Last-In, First-Out):

```
public class Stack
{
    int position;
    object[] data = new object[10]; // Can hold any type because it stores
objects

    public void Push(object obj) { data[position++] = obj; } // Pushes any
object
    public object Pop() { return data[--position]; } // Pops an object
}
```

Because `Stack` works with the `object` type, you can `Push` and `Pop` instances of *any* type:

```
Stack stack = new Stack();
stack.Push("sausage"); // Push a string
string s = (string)stack.Pop(); // Pop an object, then downcast to string
Console.WriteLine(s); // Output: sausage

stack.Push(3); // Push an integer (value type)
int three = (int)stack.Pop(); // Pop an object, then downcast to int
Console.WriteLine(three); // Output: 3
```

This ability to treat both reference types (like `string`) and value types (like `int`) as `object` instances is a core feature of C#'s **type unification**.

Boxing and Unboxing

While `object` is a **reference type** (meaning it's stored on the heap), value types (like `int`, `bool`, `structs`) can also be cast to and from `object`. This process involves special operations by the Common Language Runtime (CLR) called **boxing** and **unboxing**.

Boxing

Boxing is the process of converting a **value-type instance to a reference-type instance**. When you box a value type, the CLR wraps the value-type instance in a new `object` (or interface) on the heap.

```
int x = 9;
object obj = x; // Boxing: The value of 'x' (9) is copied into a new object on
the heap, and 'obj' references that object.
```

Unboxing

Unboxing is the reverse operation: casting an `object` (or interface) back to its original value type.

```
int y = (int)obj; // Unboxing: The value from the boxed object is copied back
into 'y'.
```

Unboxing requires an **explicit cast**. The runtime performs a strict check: the target value type *must exactly match* the actual type of the boxed object. If it doesn't, an `InvalidCastException` is thrown.

```
object obj = 9; // '9' is implicitly boxed as an 'int'
// long x = (long)obj; // InvalidCastException: The boxed object is an 'int',
not a 'long'.

// This succeeds:
object obj2 = 9;
int x2 = (int)obj2; // OK: The boxed object is an 'int', matching the target
type.

// This also succeeds, demonstrating a two-step conversion:
object obj3 = 3.5; // '3.5' is implicitly boxed as a 'double'
int x3 = (int)(double)obj3; // First, unbox to 'double', then perform a numeric
```

```
conversion to 'int'.
```

Copying Semantics: When boxing occurs, the value-type instance is *copied* into the new object on the heap. Subsequent changes to the original value-type variable do not affect the boxed copy, and vice-versa.

```
int i = 3;
object boxed = i; // 'boxed' now holds a copy of '3'
i = 5;            // 'i' changes to 5, but 'boxed' remains 3
Console.WriteLine(boxed); // Output: 3
```

While boxing and unboxing provide a unified type system, they incur a performance overhead due to the memory allocation and copying involved. For scenarios requiring high performance with collections of value types, **Generics** (which we will discuss later) offer a more efficient solution by avoiding boxing altogether.

Static and Runtime Type Checking

C# programs undergo type checking at two stages:

1. **Static Type Checking (Compile-time):** The C# compiler verifies the type correctness of your code *before* it runs. This prevents many common errors.

   ```
   // int x = "5"; // Compile-time error: Cannot implicitly convert
   string to int
   ```

2. **Runtime Type Checking (CLR-time):** The CLR performs type checking during program execution, particularly during reference conversions (downcasting) or unboxing. This ensures type safety when operations might fail based on the actual object type at runtime.

```
object y = "5";
// int z = (int)y; // Runtime error: InvalidCastException, because 'y'
actually holds a string, not an int.
```

3. Runtime type checking is possible because every object on the heap stores a "type token" that identifies its exact type.

The `GetType` Method and `typeof` Operator

To retrieve the `System.Type` object that represents a type at runtime, you have two primary mechanisms:

1. **`GetType()` Method:** This is an instance method available on all objects (inherited from `object`). It returns the `System.Type` object of the *runtime type* of the instance. `GetType()` is evaluated at runtime.

2. **`typeof` Operator:** This is a compile-time operator that takes a type name as an argument. It returns the `System.Type` object for that specific type. `typeof` is evaluated statically at compile time (or by the JIT compiler for generic type parameters).

```
public class Point { public int X, Y; }

Point p = new Point();
Console.WriteLine(p.GetType().Name);             // Output: Point
(runtime type of 'p')
Console.WriteLine(typeof(Point).Name);          // Output: Point
(compile-time type of Point)
Console.WriteLine(p.GetType() == typeof(Point)); // Output: True (both
refer to the same Type object)
Console.WriteLine(p.X.GetType().Name);           // Output: Int32
(runtime type of 'X')
Console.WriteLine(p.Y.GetType().FullName);       // Output: System.Int32
```

The System.Type class provides properties (like Name, FullName, Assembly, BaseType) and methods that expose the runtime's **reflection model**, allowing you to inspect and even manipulate types dynamically at runtime.

The ToString() Method

The ToString() method, inherited from object, provides a default textual representation of an object. All built-in types override ToString() to provide meaningful output.

```
int x = 1;
string s = x.ToString(); // s is "1"
Console.WriteLine(s);    // Output: 1
```

You can (and often should) **override** the ToString() method in your custom classes to provide a more descriptive string representation of your objects. If you don't override it, the default ToString() implementation (from object) simply returns the full name of the type.

```
public class Panda
{
    public string Name;
    public override string ToString() => Name; // Override ToString to return
the Panda's name
}

Panda p = new Panda { Name = "Petey" };
Console.WriteLine(p); // Output: Petey (calls the overridden ToString())
```

Note on boxing with ToString(): When you call an overridden object member like ToString() directly on a value type, boxing does *not* occur. Boxing only occurs if you explicitly cast the value type to object first.

```
int x = 1;
string s1 = x.ToString();    // No boxing: ToString() is called directly on
the int value.
object box = x;
string s2 = box.ToString();  // Boxing occurs: ToString() is called on the
```

```
boxed object.
```

`object` Member Listing

Here are the key members that all types inherit from `System.Object`:

```
public class Object
{
    public Object(); // Constructor
    public extern Type GetType(); // Gets the runtime Type of the instance
    public virtual bool Equals(object obj); // Compares two objects for
equality
    public static bool Equals(object objA, object objB); // Static method for
equality comparison
    public static bool ReferenceEquals(object objA, object objB); // Checks for
reference equality
    public virtual int GetHashCode(); // Returns a hash code for the object
    public virtual string ToString(); // Returns a string representation of the
object
    protected virtual void Finalize(); // Finalizer (for cleanup before garbage
collection)
    protected extern object MemberwiseClone(); // Creates a shallow copy of the
current object
}
```

The `Equals`, `ReferenceEquals`, and `GetHashCode` methods are particularly important for defining and comparing object equality, which will be covered in more detail in subsequent discussions.