A tour of the C# Language

# Introduction

At its core, C# is an ***object-oriented*** language. You define types and their behavior.

Several C# features help create robust and durable applications:

* **Garbage collection** automatically reclaims memory occupied by unreachable unused objects.
* **Nullable types** guard against variables that don't refer to allocated objects.
* **Exception handling** provides a structured and extensible approach to error detection and recovery.
* **Lambda expressions** support functional programming techniques.
* **Language Integrated Query (LINQ)** syntax creates a common pattern for working with data from any source.
* Language support for **asynchronous operations** provides syntax for building distributed systems.
* C# has a **unified type system**. All C# types, including primitive types such as int and double, inherit from a single root object type.

## .NET architecture

C# programs run on .NET, a virtual execution system called the common language runtime (CLR) and a set of class libraries.

The CLR provides other services related to automatic garbage collection, exception handling, and resource management.

## Types and variables

A *type* defines the structure and behavior of any data in C#. The declaration of a type may include its members, base type, interfaces it implements, and operations permitted for that type.

A *variable* is a label that refers to an instance of a specific type.

There are two kinds of types in C#:

* *value* types
* *reference* types.

Variables of value types directly contain their data. Variables of reference types store references to their data, the latter being known as objects.

With value types, the variables each have their own copy of the data, and it isn't possible for operations on one to affect the other (except for ref and out parameter variables).

C#'s value types are further divided into:

* simple types,
* enum types,
* struct types,
* nullable value types,
* tuple value types.

C#'s reference types are further divided into:

* class types,
* interface types,
* array types,
* delegate types.
* A class types support single inheritance and polymorphism, mechanisms whereby derived classes can extend and specialize base classes.
* A struct type is similar to a class type in that it represents a structure with data members and function members.
  + However, unlike classes, structs are value types and don't typically require heap allocation.
  + Struct types don't support user-specified inheritance, and all struct types implicitly inherit from type object.
* An interface type defines a contract as a named set of public members.

C#'s type system is unified such that a value of any type can be treated as an object.

Every type in C# directly or indirectly derives from the object class type, and object is the ultimate base class of all types.

Values of value types are treated as objects by performing boxing and unboxing operations.

int i = 123;

object o = i; // Boxing

int j = (int)o; // Unboxing

## Program structure

The key organizational concepts in C# are:

* programs,
  + Programs declare types, which contain members and can be organized into namespaces.
* namespaces,
* types,
  + Classes, structs, and interfaces are examples of types.
* members,
  + Fields, methods, properties, and events are examples of members.
* assemblies.
  + When C# programs are compiled, they're physically packaged into assemblies.
  + Assemblies typically have the file extension .exe or .dll, depending on whether they implement applications or libraries, respectively.

# Types

## Classes and objects

A class is a data structure that combines state (fields) and actions (methods and other function members) in a single unit.

The memory occupied by an object is automatically reclaimed when the object is no longer reachable. It's not necessary or possible to explicitly deallocate objects in C#.

## Structs

Classes define types that support inheritance and polymorphism. They enable you to create sophisticated behaviors based on hierarchies of derived classes.

By contrast, struct types are simpler types whose primary purpose is to store data values. Structs can't declare a base type; they implicitly derive from System.ValueType.

# C# program building blocks.

*Static methods* are accessed through the class. *Instance methods* are accessed through instances of the class.

## Static and instance methods

A method declared with a static modifier is a static method. A static method doesn't operate on a specific instance and can only directly access static members.

A method declared without a static modifier is an instance method. An instance method operates on a specific instance and can access both static and instance members.

## Virtual, override, and abstract methods

A **virtual** method is one declared and implemented in a base class where any derived class may provide a more specific implementation.

An **override** method is a method implemented in a derived class that modifies the behavior of the base class' implementation.

An **abstract** method is a method declared in a base class that must be overridden in all derived classes. In fact, abstract methods don't define an implementation in the base class.

An abstract method is declared with the abstract modifier and is permitted only in an abstract class.

# Other function members

Members that contain executable code are collectively known as the function members of a class.

Methods are the primary types of function members.

Other kinds of function members supported by C#:

* Constructors
* Properties
* Indexers
* Events
* Operators
* finalizers.

## Indexers

An **indexer** is a member that enables objects to be indexed in the same way as an array.

An indexer is declared like a property except that the name of the member is this followed by a parameter list written between the delimiters [ and ].

The parameters are available in the accessor(s) of the indexer. Similar to properties, indexers can be read-write, read-only, and write-only, and the accessor(s) of an indexer can be virtual.

## Events

An **event** is a member that enables a class or object to provide notifications.

An event is declared like a field except that the declaration includes an event keyword and the type must be a delegate type.

## Operators

An **operator** is a member that defines the meaning of applying a particular expression operator to instances of a class.

Three kinds of operators can be defined: unary operators, binary operators, and conversion operators.

All operators must be declared as public and static.

## Finalizers

A **finalizer** is a member that implements the actions required to finalize an instance of a class.

Typically, a finalizer is needed to release unmanaged resources.

The finalizer for an instance is invoked automatically during garbage collection.

The **using** statement provides a better approach to object destruction.