ChatGPT

Object-Oriented Programming (OOP) in C

C# is a fully object-oriented language, which means it uses **classes** and **objects** as the core of program structure. The four main principles of OOP are **abstraction, encapsulation, inheritance, and polymorphism** [1]. In this tutorial we explain each concept with very simple C# code examples and clear, line-by-line explanations.

Classes

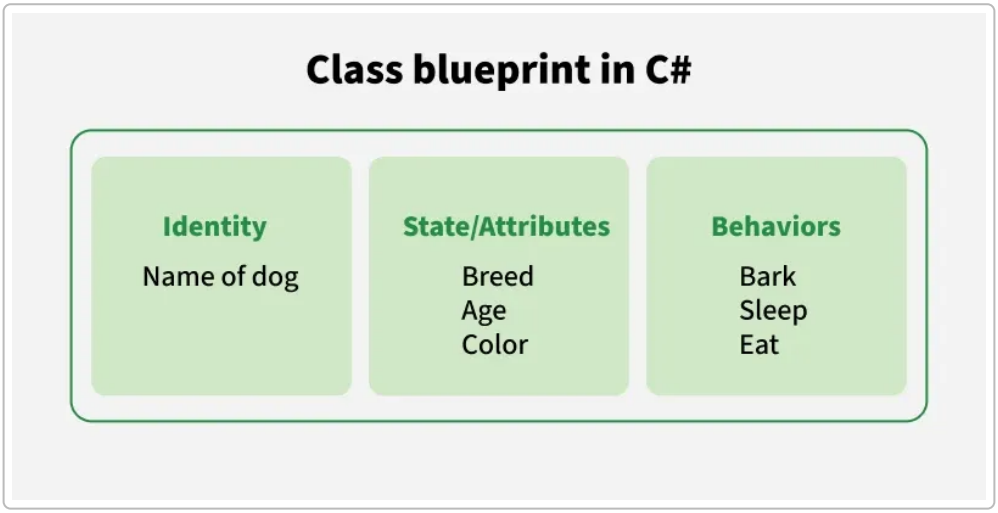

Figure: Class blueprint in C#. A class is like a blueprint for objects, defining identity, attributes (state) and behaviors (methods).

A **class** in C# is a user-defined type that groups related data and behavior [2]. In other words, a class is a blueprint from which you create objects. It defines **fields** (data) and **methods** (actions). For example, this code defines a simple `Dog` class with fields and a method:

```
public class Dog      // 1. Define a new class named Dog
{
    public string Name;  // 2. Public field (attribute) for the dog's name
    public int Age;      // 3. Public field for the dog's age

    public void Bark()   // 4. Public method (behavior) for the dog to bark
    {
        Console.WriteLine("Woof!"); // 5. When Bark() is called, print "Woof!"
```

```
    }
}
```

- **Line 1:** `public class Dog` declares a class named **Dog**. The keyword `public` means it can be accessed from other code. A class declaration groups all its contents between `{` and `}`.
- **Line 2-3:** `public string Name;` and `public int Age;` create two public fields (variables) inside the class. These define the data that each `Dog` object will hold (its name and age). Because they are `public`, other code can read or set them.
- **Line 4:** `public void Bark()` declares a method named `Bark`. Methods define behaviors or actions. This method is also `public`, so it can be called from outside the class.
- **Line 5:** Inside `Bark()`, `Console.WriteLine("Woof!");` prints a message to the console. This is the action the dog performs when we call `Bark()`.

The class itself does not occupy memory until we create an **object** (instance) from it.

Objects

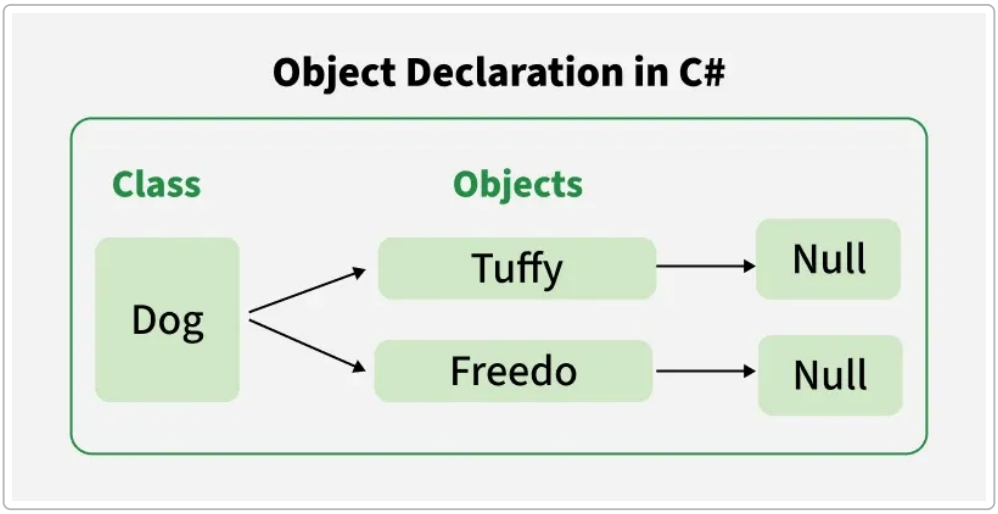

Figure: Declaring objects in C#. Here, `Dog` *is a class, and* `tuffy` *and* `freedo` *are object references (initially* `null` *before* `new`*).*

An **object** is an instance of a class. It represents a concrete entity created from the class blueprint [2]. To create (instantiate) an object in C#, use the `new` keyword. For example:

```
Dog myDog = new
Dog();   // 1. Create a new Dog object and assign it to variable myDog
myDog.Name = "Buddy";     // 2. Set the Name field of myDog to "Buddy"
myDog.Age = 5;            // 3. Set the Age field of myDog to 5

Console.WriteLine(myDog.Name + " is " + myDog.Age + " years old.");
```

```
// 4. Use the object's fields to print: "Buddy is 5 years old."
myDog.Bark();            // 5. Call the Bark() method on myDog (prints "Woof!")
```

- **Line 1:** `Dog myDog = new Dog();` declares a variable `myDog` of type `Dog` and creates a new `Dog` object. Memory is now allocated for this `Dog`, with its own `Name`, `Age`, and behaviors.
- **Lines 2-3:** We set the `Name` and `Age` fields of `myDog`. Now `myDog.Name` is `"Buddy"` and `myDog.Age` is `5`.
- **Line 4:** We access the object's fields using dot notation (`myDog.Name`, `myDog.Age`) to print a message. This reads the data from that specific object.
- **Line 5:** `myDog.Bark();` calls the `Bark` method on the `myDog` object, which causes "Woof!" to be printed.

You can create multiple objects from the same class. For example, `Dog otherDog = new Dog();` would make another dog with its own name and age. Each object has its own state, even though they share the same class definition [2].

Encapsulation (Data Hiding)

Encapsulation means bundling the data (fields) and methods that operate on the data into a single unit (the class), and **restricting direct access** to some of the object's components [3]. In practice, this is done by making fields `private` and providing `public` methods or properties to access them. This controls how the data can be set or retrieved.

For example, consider an `Account` class that encapsulates a `balance` field:

```
public class Account
{
    private double balance;            // 1. Private field: not accessible from
outside

    public void Deposit(double amount) // 2. Public method to add money
    {
        if (amount > 0)
            balance += amount;         // 3. Increase balance (validates input)
    }

    public void Withdraw(double amount)// 4. Public method to remove money
    {
        if (amount <= balance)
            balance -= amount;         // 5. Decrease balance (checks for
sufficient funds)
    }

    public double GetBalance()         // 6. Public method to read the balance
    {
        return balance;                // 7. Returns the private field's value
```

```
    }
}
```

- **Line 1:** `private double balance;` declares a private field. No code outside the `Account` class can access `balance` directly. This hides the internal state of the account.
- **Lines 2-3:** `public void Deposit(double amount)` is a public method. It allows outside code to add money. Inside, we check that the amount is positive before adding it to `balance`. This validation enforces rules on the internal data.
- **Lines 4-5:** `public void Withdraw(double amount)` similarly removes money if there is enough balance. Again, checks protect the internal state from invalid operations.
- **Lines 6-7:** `public double GetBalance()` returns the current balance. Other code calls this to read the balance.

This design hides (`private`) the `balance` field and only exposes **methods** to interact with it [4]. Thus, the internal representation (`balance`) is protected, and the class controls all access. This is encapsulation. It improves data security and maintainability [4]. (In C#, you can also use **properties** with `get`/`set` to encapsulate fields in a simpler way, but the concept is the same.)

Inheritance

Inheritance lets a class (called the *derived* or *child* class) inherit members (fields and methods) from another class (the *base* or *parent* class). This promotes code reuse and establishes a hierarchy [5]. In C#, inheritance is indicated with a colon `:`.

For example, suppose we have a base class `Animal` and a derived class `Dog`:

```
public class Animal          // Base class
{
    public void Eat()        // 1. A common method for all animals
    {
        Console.WriteLine("This animal eats food.");
    }
}

public class Dog : Animal    // Derived class inherits from Animal
{
    public void Bark()        // 2. A method specific to Dog
    {
        Console.WriteLine("The dog barks.");
    }
}
```

- `Animal` is the **base class** with a method `Eat()`.
- `Dog : Animal` defines a **derived class** `Dog` that inherits from `Animal` (colon syntax) [5].

- **Line 1:** `public void Eat()` is defined in `Animal`. All classes derived from `Animal` (like `Dog`) have this method.
- **Line 2:** `public void Bark()` is defined only in `Dog`.

Now, in `Main`, if we do:

```
Dog d = new Dog();
d.Eat();  // Inherited from Animal, prints "This animal eats food."
d.Bark(); // Defined in Dog, prints "The dog barks."
```

The `Dog` object `d` can call both `Eat()` (inherited) and `Bark()` (its own) because `Dog` inherits the `Eat` method from `Animal`. This demonstrates inheritance: derived classes automatically have the base class's members [5].

C# supports **single inheritance** (a class can inherit from only one other class directly) [6]. Inheritance establishes a clear "is-a" relationship: a `Dog` is an `Animal`.

Polymorphism

Polymorphism means "many forms". In C#, polymorphism allows the same method or action to behave differently in different contexts. There are two common types:

- **Method Overloading** (compile-time polymorphism). Multiple methods in the *same class* share the same name but have different parameter lists [7].
- **Method Overriding** (run-time polymorphism). A derived class provides a new implementation of a base class method [8].

Method Overloading

With **method overloading**, you define multiple methods with the same name in one class, as long as their parameters differ. For example:

```
public class Calculator
{
    public int Add(int a, int b)        // 1. Adds two integers
    {
        return a + b;
    }

    public int Add(int a, int b, int c) // 2. Adds three integers (same name,
different parameters)
    {
        return a + b + c;
```

```
    }
}
```

- **Line 1:** `public int Add(int a, int b)` is a method that adds two integers.
- **Line 2:** `public int Add(int a, int b, int c)` is another method **with the same name** `Add` but three parameters. This is legal because the parameter lists differ.

The compiler decides which `Add` method to call based on the number of arguments. This makes related tasks use the same method name [7]. For example:

```
Calculator calc = new Calculator();
Console.WriteLine(calc.Add(2, 3));    // Calls Add(int, int), prints 5
Console.WriteLine(calc.Add(1, 4, 5)); // Calls Add(int, int, int), prints 10
```

This is **overloading** (static polymorphism).

Method Overriding

With **method overriding**, a derived class changes (overrides) the implementation of a method defined in its base class [8]. To allow overriding, the base class method is marked `virtual`, and the derived method is marked `override`. For example:

```
public class Animal
{
    public virtual void Speak()       // 1. Virtual method in base class
    {
        Console.WriteLine("Animal makes a sound");
    }
}

public class Dog : Animal
{
    public override void Speak()      // 2. Override in derived class
    {
        Console.WriteLine("Dog says: Woof!");
    }
}
```

- **Line 1:** `public virtual void Speak()` in `Animal` means this method can be overridden by subclasses.
- **Line 2:** `public override void Speak()` in `Dog` provides a new implementation. It replaces the base behavior for `Dog` instances.

When we run:

```
Animal a1 = new Animal();
Animal a2 = new Dog(); // a Dog referenced by an Animal variable
a1.Speak();  // Prints "Animal makes a sound"
a2.Speak();  // Prints "Dog says: Woof!"
```

Here, `a2.Speak()` calls the **overridden** version in `Dog`, because at runtime the object is actually a `Dog`. This is dynamic (run-time) polymorphism, where a method call behaves differently depending on the actual object type [8].

Feature	Method Overloading	Method Overriding
Definition	Same method name, different parameters (same class) [7]	Derived class provides its own version of a base class method [8]
Polymorphism	Compile-time (static)	Run-time (dynamic)
Keywords	No special keyword needed	`virtual` (base), `override` (derived)
Example	`Add(int, int)` vs `Add(int, int, int)`	`virtual void Speak()` vs `override void Speak()`

Abstraction

Abstraction means focusing on the essential qualities of something, and hiding unnecessary details. In C#, you achieve abstraction using **abstract classes** and **interfaces** [9] [10]. Both provide a way to define *contracts* of what methods or properties a class should have, without specifying all details.

Abstract Classes

An `abstract` class is a class that cannot be instantiated on its own and may contain **abstract methods** (without a body) that must be implemented by derived classes [9]. For example:

```
public abstract class Animal
{
    public abstract void Speak();    // 1. Abstract method (no body)
    public void Sleep()              // 2. Concrete method (can have an
implementation)
    {
        Console.WriteLine("Zzz");
    }
}

public class Dog : Animal
{
    public override void Speak()     // 3. Dog provides the implementation
    {
```

```
        Console.WriteLine("Woof!");
    }
}
```

- **Line 1:** `public abstract void Speak();` declares an abstract method. It has no body here, only a signature.
- **Line 2:** `public void Sleep() { ... }` is a normal (concrete) method in `Animal`. Abstract classes can have both abstract and concrete members.
- **Line 3:** In `Dog`, `public override void Speak()` provides the required implementation of `Speak()`.

You cannot do `new Animal()` because `Animal` is abstract. You must create an instance of a concrete subclass (`Dog`). This ensures `Speak()` is defined.

Interfaces

An **interface** in C# defines a completely abstract type: it only contains method (and property) *signatures* and no implementation [10]. A class that implements an interface must provide bodies for all its members. This is another way to achieve abstraction.

For example:

```
public interface IAnimal
{
    void Speak(); // Method signature (implicitly public and abstract)
    void Run();   // Another method signature
}

public class Cat : IAnimal
{
    public void Speak()   // 1. Implement interface method Speak
    {
        Console.WriteLine("Meow");
    }
    public void Run()     // 2. Implement interface method Run
    {
        Console.WriteLine("The cat runs.");
    }
}
```

- In the `IAnimal` interface, we list `Speak()` and `Run()` but give no code.
- `Cat : IAnimal` means `Cat` must implement all methods of `IAnimal`. Lines 1-2 show those implementations.

Interfaces allow unrelated classes to share the same set of methods. You can also assign a `Cat` to an interface variable:

```
IAnimal pet = new Cat();
pet.Speak(); // Calls Cat's Speak, prints "Meow"
```

This enforces a contract: any `IAnimal` can `Speak` and `Run`, but how it does so is up to the implementing class.

Aspect	Abstract Class	Interface
Members	Can have both abstract and concrete (implemented) methods, fields, properties [9]	Only abstract method/property signatures (until C# 8) [10]
Inheritance	A class can inherit only one abstract class	A class can implement multiple interfaces
Instantiation	Cannot create instance of abstract class	Cannot create instance of interface
Use case	When classes share code and a base type	When unrelated classes share a contract

Both abstract classes and interfaces let us hide details and present only essential operations, which is the essence of abstraction [9] [10].

---

[1] Object-Oriented Programming - C# | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/tutorials/oop

[2] Class and Objects in C# - GeeksforGeeks
https://www.geeksforgeeks.org/c-sharp/class-and-object-in-c-sharp/

[3] [4] Encapsulation in C# - GeeksforGeeks
https://www.geeksforgeeks.org/c-sharp/encapsulation-in-c-sharp/

[5] [6] C# Inheritance - GeeksforGeeks
https://www.geeksforgeeks.org/c-sharp/c-sharp-inheritance/

[7] Method Overloading in C# - GeeksforGeeks
https://www.geeksforgeeks.org/c-sharp/method-overloading-in-c-sharp/

[8] Difference between Method Overriding and Method Hiding in C# - GeeksforGeeks
https://www.geeksforgeeks.org/c-sharp/difference-between-method-overriding-and-method-hiding-in-c-sharp/

[9] C# Abstraction
https://www.w3schools.com/cs/cs_abstract.php

[10] C# Interface
https://www.w3schools.com/cs/cs_interface.php