The concepts of OOP are briefly described here:

• Encapsulation is the combination of the data and actions that are related to an object. For

example, a BankAccount type might have data, such as Balance and AccountName, as well as

actions, such as Deposit and Withdraw. When encapsulating, you often want to control what

can access those actions and the data, for example, restricting how the internal state of an

object can be accessed or modified from the outside.

• Composition is about what an object is made of. For example, a Car is composed of different

parts, such as four Wheel objects, several Seat objects, and an Engine.

• Aggregation is about what can be combined with an object. For example, a Person is not part

of a Car object, but they could sit in the driver’s Seat and then become the car’s Driver—two

separate objects that are aggregated together to form a new component.

• Inheritance is about reusing code by having a subclass derive from a base or superclass. All

functionality in the base class is inherited by, and becomes available in, the derived class. For

example, the base or super Exception class has some members that have the same implemen-

tation across all exceptions, and the sub or derived SqlException class inherits those members

and has extra members that are only relevant when a SQL database exception occurs, like a

property for the database connection.

• Abstraction is about capturing the core idea of an object and ignoring the details or specifics.

C# has the abstract keyword that formalizes this concept but do not confuse the concept of

abstraction with meaning the use of the abstract keyword because it is more than that. The

concept of abstraction can also be achieved using interfaces. If a class is not explicitly abstract,

then it can be described as being concrete. Bases or superclasses are often abstract; for example,

the superclass Stream is abstract, and its subclasses, like FileStream and MemoryStream, are

concrete. Only concrete classes can be used to create objects; abstract classes can only be used

as the base for other classes because they are missing some implementation. Abstraction is

a tricky balance. If you make a class more abstract, more classes will be able to inherit from

it, but at the same time, there will be less functionality to share. A real-world example of ab-

straction is the approach car manufacturers have taken to electric vehicles (EVs). They create

a common “platform” (basically just the battery and wheels) that is an abstraction of what all

EVs need, and then add on top of that to build different vehicles like cars, trucks, vans, and so

on. The platform on its own is not a complete product, like an abstract class.

• Polymorphism is about allowing a derived class to override an inherited action to provide

custom behavior.

# Understanding stack and heap memory There are two categories of memory: stack memory and heap memory. With modern operating sys- tems, the stack and heap can be anywhere in physical or virtual memory. Stack memory is faster to work with but limited in size. It is fast because it is managed directly by the CPU and it uses a last-in, first-out mechanism, so it is more likely to have data in its L1 or L2 cache. Heap memory is slower but much more plentiful. On Windows, for ARM64, x86, and x64 machines, the default stack size is 1 MB. It is 8 MB on a typical modern Linux-based operating system. For example, in a macOS or Linux terminal, I can enter the command ulimit -a to discover that the stack size is limited to 8,192 KB and that other memory is “unlimited.” This limited amount of stack memory is why it is so easy to fill it up and get a “stack overflow.”

Defining reference and value types  
There are three C# keywords that you can use to define object types: class, record, and struct. All   
can have the same members, such as fields and methods. One difference between them is how memory is allocated:  
• When you define a type using record or class, you define a reference type. This means that the   
memory for the object itself is allocated on the heap, and only the memory address of the object   
(and a little overhead) is stored on the stack. Reference types always use a little stack memory.  
• When you define a type using record struct or struct, you define a value type. This means   
that the memory for the object itself is allocated to the stack.  
If a struct uses field types that are not of the struct type, then those fields will be stored on the heap,   
meaning the data for that object is stored in both the stack and the heap.  
These are the most common struct types:  
• Number System types: byte, sbyte, short, ushort, int, uint, long, ulong, float, double,   
and decimal  
• Other System types: char, DateTime, DateOnly, TimeOnly, and bool  
• System.Drawing types: Color, Point, PointF, Size, SizeF, Rectangle, and RectangleF  
Almost all the other types are class types, including string aka System.String and object aka   
System.Object.

# Class vs Struct

In .NET, including .NET 8, the primary differences between **structs** and **classes** are rooted in their type classification and how they are handled in memory.

## Key Differences

**Type Classification**

* **Structs** are **value types**.
* **Classes** are **reference types**.

**Memory Allocation**

* **Structs** are typically stored on the **stack**, though they can be stored on the heap if they are part of a reference type or an array of structs.
* **Classes** are stored on the **heap**, with a reference to the object stored on the stack.

**Passing to Methods**

* When passing a **struct** to a method, a **copy** of the struct is passed (call by value).
* When passing a **class** to a method, a **reference** to the class is passed (call by reference).

**Inheritance and Polymorphism**

* **Structs** cannot inherit from other structs or classes (except for implementing interfaces), and they are implicitly sealed.
* **Classes** can inherit from other classes and support polymorphism.

**Nullability**

* **Structs** cannot be null unless they are nullable structs (e.g., int?).
* **Classes** can be null.

**Constructors and Destructors**

* **Structs** have an implicit public parameterless constructor and cannot have explicit parameterless constructors or destructors.
* **Classes** can have explicit constructors and destructors.

**Garbage Collection**

* **Structs** are not subject to garbage collection since they are value types.
* **Classes** are managed by the garbage collector.

Choosing Between Structs and Classes

* **Use structs** for small, simple data types where copying is acceptable and performance is critical, such as points or colors.
* **Use classes** for more complex objects that require inheritance, polymorphism, or when the object's lifetime needs to be managed by the garbage collector.