Classes, Structures, and Namespaces

1. Classes and Objects

What is this?

A class is a data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, finalizers, and static constructors), and nested types.

How/where can I use this?

Classes can be used to create objects that represent real-world entities, such as people, animals, or things. For example, you could create a class called Person to represent a person. This class would have data members such as name, age, and address.

Example:

```
public class Person
{
    0 references
    public string Name { get; set; }
    0 references
    public int Age { get; set; }
    0 references
    public string Address { get; set; }
}
```

```
Person person = new Person();
person.Name = "Ali";
person.Age = 18;
person.Address = "Khujand";
```

2. Constructors, Initializers, and Destructors

What is this?

Constructors initialize objects, destructors clean up resources when objects are no longer needed.

How/where can I use this?

Use constructors to set up initial state, destructors for cleanup tasks.

Example:

```
public class Person
{
    2 references
    public string Name { get; set; }
    2 references
    public int Age { get; set; }
    2 references
    public Person(string name, int age) // Constructor
    {
        Name = name;
        Age = age;
    }
    0 references
    ~Person()// Destructor
    {
        // Cleanup code
    }
}
// Initializing object using constructor
Person john = new Person("John", 25);
```

3. Fields and Properties

What is this?

Fields are variables in a class/struct, properties provide controlled access to fields.

How/where can I use this?

Use fields for internal data, properties for controlled access.

Example:

```
public class Person
{
    // Field
    2 references
    private int _age;

    // Property
    1 reference
    public int Age
    {
        get { return _age; }
        set { _age = value < 18 ? 18 : value; }
    }
}
```

4. Method and static method in class

What is this?

A method is a function within a class, and a static method belongs to the class rather than an instance.

How/where can I use this?

Methods encapsulate behavior, while static methods provide functionality without creating an instance.

Example:

```
0 references
public class Calculator
{
    0 references
    public int Add(int a, int b)// Method
    {
        return a + b;
    }

    0 references
    public static double Square(double num)// Static method
    {
        return num * num;
    }
}
```

5. Structures

What is this?

Structures are value types that can contain fields, properties, and methods.

How/where can I use this?

Structures are suitable for small, simple data types that can be efficiently stored on the stack.

Example:

```
public struct Point
{
    1 reference
    public int X;
    1 reference
    public int Y;
}

// Creating an instance of the Point structure
Point p = new Point { X = 1, Y = 2 };
```

6. Record Type

What is this?

Record types provide a concise syntax for creating immutable data types.

How/where can I use this?

Records are useful for modeling data that should not change after creation.

Example:

```
public record Person(string FirstName, string LastName);

// Creating a record
Person person = new Person("John", "Doe");
```

7. Namespace and Global Namespace

What is this?

Namespaces organize code into logical groups, and the global namespace is the default namespace for a C# program.

How/where can I use this?

Namespaces prevent naming conflicts and improve code organization.

Example:

```
// Namespace declaration
namespace MyNamespace
{
    2 references
    public class MyClass { }
}

// Using a class from a specific namespace
MyNamespace.MyClass myObject = new MyNamespace.MyClass();
```

8. Partial and Extended Classes

What is this?

Partial classes allow a class to be defined in multiple files, and extension methods extend the functionality of existing classes.

How/where can I use this?

Partial classes aid code organization, and extension methods add new functionality to existing types.

Example:

```
// Partial class file 1
1 reference
public partial class MyClass
{
    0 references
    public string FirstName {get; set;}
    0 references
    public void Method1() { }
}

// Partial class file 2
1 reference
public partial class MyClass
{
    0 references
    public string LastName {get; set;}
    0 references
    public void Method2() { }
}
```

```
// Extension method
0 references
public static class StringExtensions
{
    0 references
    public static bool IsPalindrome(this string str)
    {
        // Implementation for checking palindrome
        return str.SequenceEqual(str.Reverse());
    }
}
```

8. Partial and Extended Classes

What is this?

Partial classes allow a class to be defined in multiple files, and extension methods extend the functionality of existing classes.

How/where can I use this?

Partial classes aid code organization, and extension methods add new functionality to existing types.

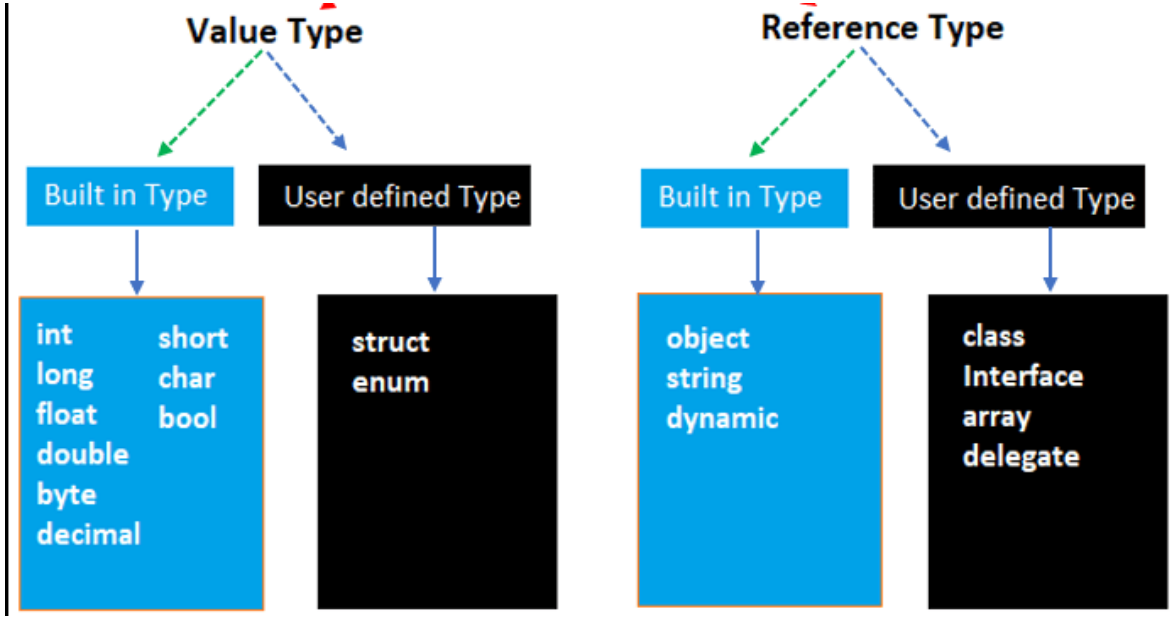
Example:

```
// Partial class file 1
1 reference
public partial class MyClass
{
    0 references
    public string FirstName {get; set;}
    0 references
    public void Method1() { }
}

// Partial class file 2
1 reference
public partial class MyClass
{
    0 references
    public string LastName {get; set;}
    0 references
    public void Method2() { }
}
```

```
// Extension method
0 references
public static class StringExtensions
{
    0 references
    public static bool IsPalindrome(this string str)
    {
        // Implementation for checking palindrome
        return str.SequenceEqual(str.Reverse());
    }
}
```

9. Value types and reference types



- While **value types are stored generally in the stack** (the stack is only 1MB for 32-bit and 4MB for 64-bit), **reference types are stored in the heap**.
- **A value type** derives from System.ValueType and contains the data inside its own memory allocation. In other words, variables or objects or value types have their own copy of the data.
- **A reference type**, meanwhile, extends System.Object and points to a location in the memory that contains the actual data. You can imagine a reference type similar to a pointer that is implicitly dereferenced when you access them.
- **String**- is not a value type since they can be huge and need to be stored on the heap. And it's **immutable** because it cannot be changed after the object is created.

10. Nullability in value types and reference types

Null in Value Types:

Value types cannot be null by default. They always have a default value (e.g., 0 for numeric types, false for bool).

You can use nullable value types (e.g., int?) to represent nullability.

Null in Reference Types:

Reference types can be null, indicating the absence of an object.

It's important to check for null before accessing members of a reference type to avoid null reference exceptions.

11. Accessibility of class and class members

Accessibility:

The accessibility of a class and its members in C# refers to the level of visibility or exposure that these elements have within different parts of your code or in other assemblies. Accessibility is specified using access modifiers. There are several access modifiers in C#, each serving a different purpose:

1. **Private**: Members with private access are accessible only within the same class.
2. **Protected**: Members with protected access are accessible within the same class and its derived classes.
3. **Internal**: Members with internal access are accessible within the same assembly (project or DLL).
4. **Public**: Members with public access are accessible from anywhere, both within the same assembly and from other assemblies.