GUI PROGRAMMING
VISUAL STUDIO C#

Graphical User Interface

- An interface that allows users to interact with electronic devices through graphical icons and visual indicators such as secondary notation as opposed to text-based interfaces, typed command labels or text navigation.
- It was introduce due to the perceived steep learning curve of the command line interfaces (CLI) which require commands to be typed in the keyboard.

GUI early developments

- Alto computer, developed by Xerox PARC in 1973 had a bitmapped screen and was the first computer to demonstrate the desktop metaphor and GUI.
- SGI 1000 series and MEX was the first graphical terminals (IRIS 1000) shipped in the late 1983.
- Macintosh released in 1984 was the first commercially successful product to use a multi-panel window interface.
  - Desktop metaphor was used in which files looks like pieces of paper, file directories looked like file folders and desk accessories i.e. calculator et al
- Windows 3 based on Common User Access (CUA), its popularity truly exploded, launched 1990
- X Window System, commonly called X11, the standard windowing system of the Unix world

Human Computer Interaction

- Involves in the study, planning, design and uses of the interaction between people (users) and computers.
- It is often regarded as the intersection of computer science, behavioral sciences, design, media studies and other fields.
- The GUI prevalence has made its use in desktop applications internet browsers, handheld computers and kiosks.
- Other UIs includes
  - Voice User Interface for speech recognition and synthesizing systems
  - Multimodal and Gestalt User Interface allow humans to engage with embodied character agents in a way that cannot be achieved with other paradigms

HCI definitions

- A discipline concerned with the design, evaluation, and implementation of interactive computing systems for human use and with the study of major phenomena surrounding them.
- Relevance on the machine side include
  - Techniques in computer graphics, OS, PL and IDEs.
- Relevance on the human side include
  - Communication theory, graphic and industrial design disciplines, linguistics, social sciences, cognitive psychology, **computer user satisfaction** et al.

HCI interests

- Methodologies and processes for designing interfaces (i.e., given a task and a class of users, design the best possible interface within given constraints, optimizing for a desired property such as learnability or efficiency of use).
- **Methods for implementing interfaces (e.g. software toolkits and libraries)**.
- Techniques for evaluating and comparing interfaces.
- Developing new interfaces and interaction techniques.
- Developing descriptive and predictive models and theories of interaction.

PL IDE setup

- Visual Studio Express 2013 for Desktop and Web may be downloaded for free from Microsoft (microsoft.com) website.
- It may be installed in Windows-based computer.
- It can be installed on top of Macintosh-based computer but with the use of virtual machine platform with (guest) OS using Windows.
  - Find and download virtual machine application for Macintosh from Vmware (vmware.com) website.
- Install C# or all of the components available with Visual Studio Express 2013 for Desktop and Web.

C# pronounced as “See Sharp”

- A simple, modern, object-oriented, and type-safe programming language.
- Has its roots in the C family of languages and will be immediately familiar to C, C++ and Java programmers.
- Is standardized as the ECMA-334 standard and ISO/IEC 23270.
- An object-oriented language but it further include support for component-oriented programming.

Programming concepts and C# fundamentals, IDE familiarization

- Writing Console Applications and learning the IDE/editor features
  - Hello World
  - Operators
  - Iteration
  - Strings
  - I/O
  - Arrays
  - Enum
  - Collection
  - Lambda

C# as component-oriented programming

- Provides language constructs to directly support components that present a programming model with
  - Properties
  - Methods and
  - Events
    - Have attributes that provide declarative information about the component and they incorporate their own documentation.
- A very natural language in which to create and use software components

C# construction of robust and durable applications

- Garbage collection, reclaims memory occupied by unused objects.
- Exception handling, provides a structured and extensible approach to error detection and recovery.
- Type-safe, makes it impossible to read from uninitialized variables, to index arrays beyond their bounds, or to perform unchecked type casts.

C# unified type system

- Primitive types such as `int` and `double` inherit from a single root `object` type.
- All types share a set of common operations and values of any type can be stored, transported and operated upon in a consistent manner.
- Supports both user-defined reference types and value types, allowing dynamic allocation of objects as well as in-line storage of lightweight structures.

C# versioning and design

- Ensures programs and libraries can evolve over time in a compatible manner.
- Other PL's new version and dependent libraries when introduce breaks programs/applications more often than necessary.
- Aspects
    - Separation of `virtual` and `override` modifiers.

C# program introduction

- A source file typically have file extension `.cs`.
- A source file when compiled produces an executable assembly named `.exe` file extension.
- The program starts with a `using` directive that reference the `System` namespace
- For example `System` namespace contains a number of types such as the
  - `Console` class referenced in the program,
  - Other namespaces such as `IO` and `Collections`.

Key organizational concepts

- Programs
- Namespaces
- Types
- Members
- Assemblies

Programs

- Consist of one or more source files.
- Declare types, which contain members and can be organized into namespaces.
    - Classes and interfaces are examples of types.

Namespaces

- Provides a hierarchical means of organizing C# programs and libraries.
- Are used both as an ;internal′ organization system for a system and as an external organization system—a way of presenting elements that are exposed to other programs.
- Are implicitly public and the declaration of a namespace cannot include any access modifiers.

Namespaces include compilation unit

- Defines the overall of a source file. A compilation unit consists of zero or more *using-directive*s followed by zero or more *global-attributes* followed by zero or more *namespace-member-declaration*s.

```
compilation-unit:
extern-alias-directives_opt   using-directives_opt  global-attributes_opt
                namespace-member-declarations_opt
```

- Consists of one or more compilation units, each contained in a separate source file. When a program is compiled, all of the compilation units are processed together. Thus, compilation units can depend on each other, possibly in a circular fashion.

Namespaces include compilation unit continuation

- The *using-directives* of a compilation unit affect the global-attributes and namespace-member-declarations of that compilation unit, but have no effect on other compilation units.
- The *global-attributes* of a compilation unit permit the specification of attributes for the target assembly and module. Assemblies and modules act as physical containers for types. An assembly may consist of several physically separate modules.
- The *namespace-member-declarations* of each compilation unit of a program contribute members to a single declaration space called the *global namespace*.

Namespace include namespace declarations

- Consists of the keyword namespace, followed by a namespace name and body, optionally followed by a semicolon.
- May occur as a top-level declaration in a *compilation-unit* or as a member declaration within another *namespace-declaration*.

namespace-declaration:
namespace   *qualified-identifier*   *namespace-body*   ;$_{opt}$

qualified-identifier:
identifier
qualified-identifier   .   *identifier*

namespace-body:
{   *extern-alias-directives*$_{opt}$ *using-directives*$_{opt}$ *namespace-member-declarations*$_{opt}$ }

Namespace include extern aliases

- introduces an identifier that serves as an alias for a namespace. The specification of the aliased namespace is external to the source code of the program and applies also to nested namespaces of the aliased namespace.
- The scope of an *extern-alias-directive* extends over the *using-directives, global-attributes* and *namespace-member-declarations* of its immediately containing compilation unit or namespace body.
- Within a compilation unit or namespace body that contains an *extern-alias-directive,* the identifier introduced by the *extern-alias-directive* can be used to reference the aliased namespace. It is a compile-time error for the *identifier* to be the word `global`.

Namespace include extern aliases continuation

- Makes an alias available within a particular compilation unit or namespace body, but it does not contribute any new members to the underlying declaration space. In other words, an *extern-alias-directive* is not transitive, but, rather, affects only the compilation unit or namespace body in which it occurs.

Namespace include using directives

- It facilitates the use of namespaces and types defined in other namespaces.
- It impact the name resolution process of namespace-or-type-names and simple-names, but unlike declarations, using directives do not contribute new members to the underlying declaration spaces of the compilation units or namespaces within which they are used.

using-directives:

  using-directive

  using-directives   using-directive

using-directive:

  using-alias-directive

  using-namespace-directive

Namespace include using directives continuation

- *using-alias-directive* introduces an alias for a namespace or type.
- *using-namespace-directive* imports the type members of a namespace.
- The scope of a *using-directive* extends over the namespace-member-declarations of its immediately containing compilation unit or namespace body. The scope of a *using-directive* specifically does not include its peer using-directives. Thus, peer *using-directives* do not affect each other, and the order in which they are written is insignificant.

Namespace include namespace members

- A *namespace-member-declaration* is either a *namespace-declaration* or a *type-declaration*.

namespace-member-declarations:

  namespace-member-declaration

  namespace-member-declarations   namespace-member-declaration

namespace-member-declaration:

  namespace-declaration

  type-declaration

- A compilation unit or a namespace body can contain namespace-member-declarations, and such declarations contribute new members to the underlying declaration space of the containing compilation unit or namespace body.

Namespace include type declaration

- A type-declaration is a class-declaration, a struct-declaration, an interface-declaration, an enum-declaration, or a delegate-declaration.

type-declaration:

```
        class-declaration
        struct-declaration
        interface-declaration
        enum-declaration
        delegate-declaration
```

- A type-declaration can occur as a top-level declaration in a compilation unit or as a member declaration within a namespace, class, or struct.

Namespace include type declaration continuation

- The permitted access modifiers and the default access for a type declaration depend on the context in which the declaration takes place:
  - Types declared in compilation units or namespaces can have public or internal access. The default is internal access.
  - Types declared in classes can have public, protected internal, protected, internal, or private access. The default is private access.
  - Types declared in structs can have public, internal, or private access. The default is private access.

Namespace include namespace alias qualifiers

- The ***namespace alias qualifier*** **::** makes it possible to guarantee that type name lookups are unaffected by the introduction of new types and members. The namespace alias qualifier always appears between two identifiers referred to as the left-hand and right-hand identifiers. Unlike the regular . qualifier, the left-hand identifier of the **::** qualifier is looked up only as an extern or using alias.
- A *qualified-alias-member* is defined as follows:

qualified-alias-member:

identifier **::** *identifier* *type-argument-list*$_{opt}$

Namespace include namespace alias qualifiers continuation

- A *qualified-alias-member* can be used as a *namespace-or-type-name* or as the left operand in a member-access.
- A *qualified-alias-member* has one of two forms:
    - N::I<A1, ..., AK>, where N and I represent identifiers, and <A1, ..., AK> is a type argument list. (K is always at least one.)
    - N::I, where N and I represent identifiers. (In this case, K is considered to be zero.)

Types

- Examples
    - Classes
    - Interfaces

Type class

- A data structure that may contain data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors and static constructors), and nested types. Class types support inheritance, a mechanism whereby a derived class can extend and specialize a base class.

Type class continuation

- A class-declaration is a type-declaration that declares a new class.

class-declaration:

attributesopt class-modifiersopt partialopt class identifier type-parameter-listopt class-baseopt type-parameter-constraints-clausesopt class-body ;opt

- A class-declaration consists of an optional set of attributes, followed by an optional set of class-modifiers, followed by an optional partial modifier, followed by the keyword class and an identifier that names the class, followed by an optional type-parameter-list, followed by an optional class-base specification, followed by an optional set of type-parameter-constraints-clauses, followed by a class-body, optionally followed by a semicolon.

Type class modifiers

class-modifiers:
  class-modifier
  class-modifiers   *class-modifier*

class-modifier:
  new
  public
  protected
  internal
  private
  abstract
  sealed
  static

Type class modifiers

- The public, protected, internal, and private modifiers control the accessibility of the class. Depending on the context in which the class declaration occurs, some of these modifiers may not be permitted.
- The **abstract**, **sealed** and **static** modifiers are discussed in the following sections.

Type class abstract modifier

- It is used to indicate that a class is incomplete and that it is intended to be used only as a base class. An abstract class differs from a non-abstract class in the following ways:
  - An abstract class cannot be instantiated directly, and it is a compile-time error to use the new operator on an abstract class. While it is possible to have variables and values whose compile-time types are abstract, such variables and values will necessarily either be null or contain references to instances of non-abstract classes derived from the abstract types.
  - An abstract class is permitted (but not required) to contain abstract members.
  - An abstract class cannot be sealed.

Type class sealed modifier

- It is used to prevent derivation from a class. A compile-time error occurs if a sealed class is specified as the base class of another class.
- A sealed class cannot also be an abstract class.
- The sealed modifier is primarily used to prevent unintended derivation, but it also enables certain run-time optimizations. In particular, because a sealed class is known to never have any derived classes, it is possible to transform virtual function member invocations on sealed class instances into non-virtual invocations.

Type class static modifier

- It is used to mark the class being declared as a static class.
- It cannot be instantiated, cannot be used as a type.
- It can contain only static members. Only a static class can contain declarations of extension methods.

Type class static modifier continuation

- A static class declaration is subject to the following restrictions:
  - A static class may not include a sealed or abstract modifier. Note, however, that since a static class cannot be instantiated or derived from, it behaves as if it was both sealed and abstract.
  - A static class may not include a class-base specification and cannot explicitly specify a base class or a list of implemented interfaces. A static class implicitly inherits from type object.
  - A static class can only contain static members. Note that constants and nested types are classified as static members.
  - A static class cannot have members with protected or protected internal declared accessibility.

Members of a class type

- Constants
- Fields
- Methods
- Properties
- Events
- Indexers
- Operators
- Instance constructors
- Destructors
- Static constructors
- Types

Members of a class type, constants

- Represent constant values associated with the class.
- A value that can be computer at compile time.
- A constant declaration that declares multiple constants is equivalent to multiple declarations of single constants with the same attributes, modifiers, and type. For example

```
class A
{
        public const double X = 1.0, Y = 2.0, Z = 3.0;
}
```

is equivalent to

```
class A
{
        public const double X = 1.0;
        public const double Y = 2.0;
        public const double Z = 3.0;
}
```

Members of a class type, constants

- The type specified in a constant declaration must be sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, an enum-type, or a reference-type.

Members of a class type, fields

- The variables associated with the class or an object.
- There are also the following:
    - Static and instance fields
    - Read-only fields
    - Volatile fields
    - Field initialization
    - Variable initializers

Members of a class type, fields

- The variables associated with the class or an object.
- A field declaration that declares multiple fields is equivalent to multiple declarations of single fields with the same attributes, modifiers, and type. For example

```
class A
{
      public static int X = 1, Y, Z = 100;
}
```

is equivalent to

```
class A
{
      public static int X = 1;
      public static int Y;
      public static int Z = 100;
}
```

Members of a class type, methods

- Implements the computations and actions that can be performed by the class or an object.
- A declaration has a valid combination of modifiers if all of the following are true:
  - The declaration includes a valid combination of access modifiers.
  - The declaration does not include the same modifier multiple times.
  - The declaration includes at most one of the following modifiers: static, virtual, and override.
  - The declaration includes at most one of the following modifiers: new and override.
  - If the declaration includes the abstract modifier, then the declaration does not include any of the following modifiers: static, virtual, sealed or extern.
  - If the declaration includes the private modifier, then the declaration does not include any of the following modifiers: virtual, override, or abstract.
  - If the declaration includes the sealed modifier, then the declaration also includes the override modifier.
  - If the declaration includes the partial modifier, then it does not include any of the following modifiers: new, public, protected, internal, private, virtual, sealed, override, abstract, or extern.

Members of a class type, static and instance methods

- When a method declaration includes a static modifier, that method is said to be a static method. When no static modifier is present, the method is said to be an instance method.
- A static method does not operate on a specific instance, and it is a compile-time error to refer to this in a static method.

Members of a class type, virtual methods

- When an instance method declaration includes a virtual modifier, that method is said to be a virtual method. When no virtual modifier is present, the method is said to be a non-virtual method.
- The implementation of a non-virtual method is invariant: The implementation is the same whether the method is invoked on an instance of the class in which it is declared or an instance of a derived class. In contrast, the implementation of a virtual method can be superseded by derived classes. The process of superseding the implementation of an inherited virtual method is known as overriding that method

Members of a class type, override methods

- When an instance method declaration includes an override modifier, the method is said to be an override method. An override method overrides an inherited virtual method with the same signature. Whereas a virtual method declaration introduces a new method, an override method declaration specializes an existing inherited virtual method by providing a new implementation of that method.
- The method overridden by an override declaration is known as the overridden base method.

Members of a class type, override methods continuation

- A compile-time error occurs unless all of the following are true for an override declaration:
  - An overridden base method can be located as described above.
  - There is exactly one such overridden base method. This restriction has effect only if the base class type is a constructed type where the substitution of type arguments makes the signature of two methods the same.
  - The overridden base method is a virtual, abstract, or override method. In other words, the overridden base method cannot be static or non-virtual.
  - The overridden base method is not a sealed method.
  - The override method and the overridden base method have the same return type.
  - The override declaration and the overridden base method have the same declared accessibility. In other words, an override declaration cannot change the accessibility of the virtual method. However, if the overridden base method is protected internal and it is declared in a different assembly than the assembly containing the override method then the override method's declared accessibility must be protected.
  - The override declaration does not specify type-parameter-constraints-clauses. Instead the constraints are inherited from the overridden base method. Note that constraints that are type parameters in the overridden method may be replaced by type arguments in the inherited constraint. This can lead to constraints that are not legal when explicitly specified, such as value types or sealed types.

Members of a class type, sealed methods

- When an instance method declaration includes a sealed modifier, that method is said to be a ***sealed method***.
- If an instance method declaration includes the sealed modifier, it must also include the override modifier.
- Use of the sealed modifier prevents a derived class from further overriding the method.

Members of a class type, external methods

- When an instance method declaration includes an abstract modifier, that method is said to be an ***abstract method***. Although an abstract method is implicitly also a virtual method, it cannot have the modifier virtual.
- An abstract method declaration introduces a new virtual method but does not provide an implementation of that method.

Members of a class type, external methods

- When a method declaration includes an `extern` modifier, that method is said to be an ***external method***. External methods are implemented externally, typically using a language other than C#. Because an external method declaration provides no actual implementation, the *method-body* of an external method simply consists of a semicolon. An external method may not be generic.
- The extern modifier is typically used in conjunction with a `DllImport` attribute, allowing external methods to be implemented by DLLs (Dynamic Link Libraries). The execution environment may support other mechanisms whereby implementations of external methods can be provided.

Members of a class type, external methods example

- This example demonstrates the use of the extern modifier and the DllImport attribute:

```
using System.Text;
using System.Security.Permissions;
using System.Runtime.InteropServices;
class Path {
        [DllImport("kernel32", SetLastError=true)]
        static extern bool CreateDirectory(string name, SecurityAttribute sa);
        [DllImport("kernel32", SetLastError=true)]
        static extern bool RemoveDirectory(string name);
        [DllImport("kernel32", SetLastError=true)]
        static extern int GetCurrentDirectory(int bufSize, StringBuilder buf);
        [DllImport("kernel32", SetLastError=true)]
        static extern bool SetCurrentDirectory(string name);
}
```

Members of a class type, external methods

- When a method declaration includes a partial modifier, that method is said to be a partial method. Partial methods can only be declared as members of partial types, and are subject to a number of restrictions.

Members of a class type, extension methods

- When the first parameter of a method includes the this modifier, that method is said to be an extension method. Extension methods can only be declared in non-generic, non-nested static classes. The first parameter of an extension method can have no modifiers other than this, and the parameter type cannot be a pointer type.

Members of a class type, properties

- Define named characteristics and the actions associated with reading and writing those characteristics.

Members of a class type, events

- Defines notifications that can be generated by the class.

Members of a class type, indexers

- Permit instances of the class to be indexed in the same way (syntactically) as arrays.

Members of a class type, operators

- Define the expression operators that can be applied to instances of the class.

Members of a class type, instance constructors

- Implement the actions required to initialize instances of the class.

Members of a class type, destructors

- Implements the actions to be performed before instances of the class are permanently discarded.

Members of a class type, static constructors

- Implements the actions required to initialize the class itself.

Members of a class type, types

- Represents the types that are local to the class.

Assemblies

- When programs are compiled its means they are physically packaged.
- Have file extension `.exe` or `.dll` depending on whether they implement applications or libraries.
- Contain executable code in the form of Intermediate Language instructions, and symbolic information in the form of metadata.

2 kinds of types

- Value types
  - Variables here directly contain their data.
- Reference types
  - Variables here store references to their data also known as objects.
  - It is possible for 2 variables to reference the same object and thus possible for operations on one variable to affect the object referenced by the other variable.

8 integral type system

Category		Description
Value types	Simple types	Signed integral: sbyte, short, int, long
		Unsigned integral: byte, ushort, uint, ulong
		Unicode characters: char
		IEEE floating point: float, double
		High-precision decimal: decimal
		Boolean: bool
	Enum types	User-defined types of the form enum E {...}
	Struct types	User-defined types of the form struct S {...}
	Nullable types	Extensions of all other value types with a null value
Reference types	Class types	Ultimate base class of all other types: object
		Unicode strings: string
		User-defined types of the form class C {...}
	Interface types	User-defined types of the form interface I {...}
	Array types	Single- and multi-dimensional, for example, int[] and int[,]
	Delegate types	User-defined types of the form e.g. delegate int D(...)

8 integral type system supports

- 8-bit, 16-bit, 32-bit, 64-bit
- In `signed` and `unsigned` form.
- 2 floating types, `float` and `double`, are represented using 32-bit single precision and 64-bit double precision IEEE 754 formats.
- `decimal` type is a 128-bit data type suitable for monetary and financial calculations.
- `bool` type is used to represent Boolean values—that are either `true` or `false`.
- Unicode encoding is used in processing of character (`char`) and string (`string`), each of which represents a UTF-16 code units.
  - Universal Character Set Transformation Format
    - UTF-8 is the dominant character encoding for the WWW

UTF-8 sample for new HTML file using WebMatrix

```
<!DOCTYPE html>

<html lang="en">
    <head>
        <meta charset="utf-8" />
        <title></title>
    </head>
    <body>

    </body>
</html>
```

Numeric types

Category	Bits	Type	Range/Precision
Signed integral	8	sbyte	–128...127
	16	short	–32,768...32,767
	32	Int	–2,147,483,648...2,147,483,647
	64	Long	–9,223,372,036,854,775,808...9,223,372,036,854,775,807
Unsigned integral	8	byte	0...255
	16	ushort	0...65,535
	32	uint	0...4,294,967,295
	64	ulong	0...18,446,744,073,709,551,615
Floating point	32	float	1.5×10^{-45} to 3.4×10^{38}, 7-digit precision
	64	double	5.0×10^{-324} to 1.7×10^{308}, 15-digit precision
Decimal	128	decimal	1.0×10^{-28} to 7.9×10^{28}, 28-digit precision

Creating new types

- Type declarations is used
    - Specifies the name and the members of the new type.

Categories of types, user-definable

- Class types,
- Struct types,
- Interface types,
- Enum types, and
- Delegate types.

Class types defines data structure containing

- Data members (fields),
- Function members (methods, properties and others).

- Supports single inheritance and polymorphism
  - Mechanisms whereby derived classes can extend and specialize base classes.

Struct type

- Similar to class type.
- It represents a structure with data members and function members.
- Do not require heap allocation (unlike classes).
- Do not support user-specified inheritance.
- Implicitly inherit from type `object`.

Interface type defines

- A contract as a named set of public function members.
- A class or struct that implements an interface must provide implementations of the interface's function members.
- An interface may inherit from multiple base interfaces, and a class or struct may implement multiple interfaces.

Delegate type represents

- References to methods with a particular parameter list and return type.
- Delegates make it possible to treat methods as entities that can be assigned to variables and passed as parameters.
- Delegates are similar to the concept of function pointers found in some other languages, but unlike function pointers, delegates are object-oriented and type-safe.

Types that support generics, can be parameterized with other types

- Class
- Struct
- Interface
- Delegate

Enum type

- A distinct type with named constants.
- Has underlying type
  - Must be one of the eight integral types.
- Its set of values is the same as the set of values of the underlying type.

Single- and multi-dimensional array types

- Do not need to be declared before they can be used.
- Are constructed by following a type name with square brackets.
- Examples
  - `int[ ]` is a single-dimensional array of int,
  - `int[ , ]` is a two-dimensional array of int, and
  - `int[ ][ ]` is a single-dimensional array of single-dimensional arrays of int.

Nullable types

- Do not have to be declared before they can be used.

`object` class type

- Every type, directly or indirectly, were derived from this class.
- Values of reference types are treated as objects simply by viewing the values as type object.
- Values of value types are treated as objects by performing ***boxing*** and ***unboxing*** operations, example

```
using System;
class Test {
      static void Main() {
            int i = 123;
            object o = i;       // Boxing
            int j = (int)o;     // Unboxing
      }
}
```

C# unified type system

- Effectively means that value types can become objects “on demand.”
- The unification, general-purpose libraries that use type object can be used with both reference types and value types.

Variables include

- Fields
- Array elements
- Local variables and
- Parameters.

- It represents storage locations
  - Every variable has a type of that determines what values can be stored in the variable.

Variable type

Type of Variable	**Possible Contents**
Non-nullable value type	A value of that exact type
Nullable value type	A null value or a value of that exact type
`object`	A null reference, a reference to an object of any reference type, or a reference to a boxed value of any value type
Class type	A null reference, a reference to an instance of that class type, or a reference to an instance of a class derived from that class type
Interface type	A null reference, a reference to an instance of a class type that implements that interface type, or a reference to a boxed value of a value type that implements that interface type
Array type	A null reference, a reference to an instance of that array type, or a reference to an instance of a compatible array type
Delegate type	A null reference or a reference to an instance of that delegate type