Nombre y Apellido\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Matricula\_\_\_\_\_\_\_\_\_\_

**Ejercicio 1** Realice un programa de su elección utilizando el lenguaje de programación de su elección donde se incluyan las sentencias “**IF**”, “**Else-IF**”, “**For**” y “**Case**” las variables “**Int**”, “**Decimal**”, “**String**” y “**Booblean**” y se utilice cuando menos 1 función.

**Ejercicio 2** Responda Correctamente.

1. ¿Cuáles son las ventajas de C# sobre C, C++ o Java?

Como C++ y Java, C# es un lenguaje de programación de alto nivel y orientado a objetos. Es normalmente más eficiente que Java y tiene características útiles como la sobrecarga de operadores. C# está basado en C++ pero tiene varias ventajas sobre este lenguaje más antiguo: tiene tipado seguro, orientación a objetos más comprensiva, y la sintaxis ha sido simplificada en importantes formas. Y lo más importante, C# interactúa excepcionalmente bien con otros lenguajes de la plataforma .NET. Por esta razón, C# es una mejor opción para construir aplicaciones para .NET.

2. ¿Cómo se utilizan los nombres de espacio en C#?

Las Clases del framework .NET pueden ser organizadas usando nombres de espacio. El ámbito de una clase se declara usando la palabra clave de nombre de espacio. Entonces puede incluir métodos del nombre de espacio en su código incluyendo la línea “using [namespace];” al principio de su programa.

3. ¿Qué es un constructor?

Un constructor es el método de una clase que se llama cuando un objeto de esa clase se crea. El constructor inicializa parámetros de la clase y tiene el mismo nombre que la clase.

4. ¿Qué es un destructor?

Un destructor elimina un objeto de clase de la memoria. Se llama cuando el objeto es explícitamente eliminado por el código que escribe, o cuando el objeto sale del ámbito, lo que puede suceder cuando el programa sale de una función. El destructor tiene el mismo nombre que la clase pero precedido por una tilde (~).

5. ¿Cómo se sobrecargan métodos en C#?

Puede sobrecargar métodos en C# especificando un número diferente de parámetros en la definición del método. La sobrecarga puede ayudar a dar a su programa la flexibilidad que necesita para operar con diferentes tipos de entrada de datos.

6. ¿Por qué usar la encapsulación?

La encapsulación – combinar definiciones de funciones y datos juntos en un clase – se usa para separar partes de código del resto del programa. Esto permite que los datos privados de un objeto permanezcan ocultos del resto del programa, mantener el código limpio y fácil de entender y permitir que las clase puedan ser reutilizadas en otros programas.

7. ¿Cuál es la diferencia entre una clase y un registro (struct)?

Mientras que las clases son pasadas por referencia, los registros son pasados por valor. Las clases pueden ser heredadas, pero los registros no. Los registros generalmente dan un mejor rendimiento ya que se almacenan en la pila en lugar del montículo.

8. ¿Qué es el GAC?

El acrónimo GAC significa Caché de Ensamblados Global. El GAC es donde los ensamblados se almacenan para que muchas aplicaciones diferentes puedan compartir estos ensamblados. Múltiples versiones de los ensamblados pueden ser guardadas en el GAC, y las aplicaciones pueden especificar qué versión quieren usar en el archivo de configuración.

9. ¿Cómo ayuda .NET a gestionar las DLLs en un sistema?

Cuando tiene varias DLLs en un sistema, está en lo que se conoce como “Infierno DLL”. Gestionar las DLLs puede ser particularmente difícil si existen múltiples versiones de las distintas DLLs. En el framework .NET, los ensamblados se gestionan utilizando la información guardada en sus metadatos, y puede guardar varias versiones de cada uno en el GAC.

10. ¿Qué tipos de errores pueden ocurrir en un programa C#?

Los tres tipos de errores posibles en C# son los siguientes:

Error de Sintaxis. Este tipo de error, que se identifica durante la compilación, ocurre porque el programador ha usado una sintaxis incorrectamente o ha cometido una falta en el código.

Error de Lógica. Este tipo de error provoca que el programa realice algo distinto a lo que el programador pretendía. El programa devolverá un resultado inesperado en respuesta a las pruebas.

Error en Tiempo de Ejecución. Este tipo de error provoca que el programa aborte o termine incorrectamente.

Si puede manejar estas preguntas, tiene buenas probabilidades de obtener un trabajo como programador C#. Si ha tenido dificultades con alguna de ellas, quizás necesite refrescar sus habilidades de programación C#.

**¿Para que sirve el método Finalize en .NET?**

El recolector de basura “.NET Garbage Collector” hace casi todo para limpiar la actividad de tus objetos. Recursos no administrados (unmanaged) como por ejemplo Windows API, archivos, conexiones de base de datos, objetos COM, etc. están fuera del alcance del .NET framework. Tenemos que explicitamente limpiar nuestros recursos. Para esos tipos de objetos el .NET framework provee el método Object.Finalize el cual puede ser modificado y poner código de limpieza para recursos no administrados.

**¿Por qué se prefiere no utilizar Finalize para limpiar?**

El problema con Finalize es que la recolección de basura tiene que hacer dos rondas para remover los objetos que tienen el método Finalize. Por eso es bueno no escribir código para recursos .NET en el método Finalize en vez usar el método Dispose.

**¿Qué es el método Dispose?**

El método Dispose pertenece a la interfaz “IDisposable”. Hemos visto en la pregunta previa cuan malo puede ser utilizar el método Finalize para escribir código de limpieza para recursos no administrados. Si algún objeto quiere liberar su código no administrado, lo mejor es implementar la interfaz IDisposable y modificar el método Dispose. Una vez que tu clase ha expuesto el método Dispose, es responsabilidad del cliente llamar al método Dispose para hacer la limpieza.

**¿Cómo fuerzo la llamada al método Dispose ya que los clientes podrían olvidar llamar el método?**

Haz la llamada a Dispose en el método Finalize, en el método Dispose suprime el método Finalize usando GC.SuppressFinalize. Esta es la mejor manera de limpiar recursos no administrados:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | public class CleanClass : IDisposable  {     public void Dispose()     {        GC.SuppressFinalize(this);     }     protected override void Finalize()     {        Dispose();     }  } |

**¿Qué es una interfaz y qué es una clase abstracta?**

Respuesta 1.

En una interfaz, todos los métodos son abstractos, sin implementación. Mientras en una clase abstracta algunos métodos pueden ser concretos. En una interfaz no están permitidos los modificadores de acceso. Las interfaces y las clases abstractas son básicamente un conjunto de reglas que tienes que seguir en caso quieras usarlas (heredándolas).

Respuesta 2.

Las clases abstractas están relacionadas con las interfaces. Son clases que no pueden ser instanciadas y son frecuentemente partialmente implementadas o no implementadas. Una diferencia clave entre clases abstractas e interfaces es que una clase puede implementar un ilimitado número de interfaces pero puede heredar sólo de una clase (abstracta o de cualquier otra clase). Una clase que está derivada de “abstract” puede aún implementar interfaces. Las clases abstractas son útiles cuando se crean componentes porque ellas te permiten especificar un nivel que no varía de funcionalidad en algunos métodos pero dejan la implementación de otros métodos hasta que una específica implementación de la clase sea necesaria. Ellas favorecen el versionamiento porque si una funcionalidad adicional es necesaria en las clases derivadas, puede ser agregada a la clase base sin romper el código.

Respuesta 3.

Una clase abstracta es la única que no puede ser usada para crear objetos. Una clase abstracta está diseñada para actuar como una clase base (para ser heredada por otras clases). La clase abstracta es un concepto de diseño en el desarrollo de programas y provee una base sobre la cual otras clases son construídas. Las clases abstractas son similares a las interfaces. Luego de declarar una clase abstracta, no puede ser instanciada, debe ser heredada. Como las interfaces, las clases abstractas pueden especificar miembros que deben ser implementados en las clases que hereden de ellas. A diferencia de las interfaces, una clase puede heredar solamente una clase. Las clases abstractas pueden solamente especificar miembros que deberían ser implementados por todas las clases que hereden de ella.

Respuesta 4.

Una interfaz luce como una clase pero no tiene implementación. Son grandiosas para poner juntas arquitecturas como plug-and-play donde los componentes pueden ser intercambiados a voluntad. Piensa en la implementación de las extensiones plug-in de Firefox. Si necesitas cambiar el diseño, crea una interfaz. Podrías tener clases abstractas que proveen algún comportamiento por defecto. Las clases abstractas son excelentes candidatos dentro de los fremaworks de aplicaciones.

Respuesta 5.

Una diferencia clave entre interfaces y clases abstractas (posiblemente la más importante) es que múltiples interfaces pueden ser implementadas por una clase pero sólo una clase abstracta puede ser heredada por una sola clase.

C++ soporta herencia múltiple pero C# no. Herencia múltiple en C++ ha sido siempre controversial porque la resolución de múltiples implementaciones heredadas del mismo método de diferentes clases base es difícil de controlar y anticipar.

C# decidió evitar este problema permitiendo a una clase implementar múltiples interfaces, las cuales no contienen implementaciones de métodos pero restringiendo una clase a tener sólo una clase padre. Aunque esto puede resultar en implementaciones redundantes del mismo método cuando clases diferentes implementan la misma interfaz, es aún un excelente compromiso.

Otra diferencia entre interfaces y clases abstractas es que una interfaz puedes ser implementada por una clase abstracta, pero ninguna clase abstracta o no puede ser heredada por una interfaz.

Respuesta 6

¿Qué es una clase abstracta?

Una clase abstracta es un clase especial que no puede ser instanciada. La pregunta es ¿por qué necesitamos una clase que no puede ser instanciada? Una clase abstracta existe solamente para que pueda ser heredada. En otras palabras, solamente permite que otras clases hereden de ella pero no puede ser instanciada. La ventaja es que fuerza ciertas jerarquías para todas las sub-clases. En palabras simples, es una clase de contrato que fuerza a todas las sub-clases a llevar la misma jererquía o estándares.

¿Qué es una interfaz?

Una interfaz no es una clase, es una entidad que es definida por la palabra interfaz. Una interfaz no tiene implementación; solamente tiene la firma o en otras palabras, sólo la definición de los métodos sin el cuerpo. Como una de las similitudes con la clase abstracta, es un contrato que es utilizado para definir jerarquías para todas las sub-clases o define un conjunto específico de métodos y sus argumentos. La principal diferencia entre ellas es que una clase puede implementar más de una interfaz pero puede sólo heredar de una clase abstracta. Desde que C# no soporta herencia múltiple, las interfaces son utilizadas para implementar herencia múltiple.

**¿Cómo funciona la caché de salida (output caching) en ASP.NET?**

Es una poderosa técnica que incrementa el rendimiento de la petición/respuesta (request/response) haciendo caché al contenido generado por páginas dinámicas. Output caching está activado por defecto pero la salida de cualquier respuesta dada no es ingresada al caché a menos que una acción explícita sea tomada para hacer que esa respuesta se encuentre disponible en la caché.

Para que una respuesta sea elegible para output caching, debe tener una válida política de expiración/validación y visibilidad en la caché pública. Esto puede hacerse usando ya sea la API de bajo nivel OutputCache o la directiva de alto nivel @OutputCache. Cuando output caching está activado, una entre en la caché de salida es creada en la primera petición GET a la página. Siguientes peticiones GET o HEAD son servidas desde la entrada de la output cache hasta que la petición en caché expire.

La caché de salida también soporta variaciones de pares nombre/valor GET o POST.

La caché de salida respeta la expiración y las políticas de validación para las páginas. Si una página está en la caché de salida y has sido marcada con una política de expiración que indica que la página expira en 60 minutos del tiempo en que se almacena en caché, la página es quitada de la caché de salida luego de 60 minutos. Si otra petición es recivida luego de ese tiempo, el código de la página es ejecutado y la página puede ser almacenada en caché de nuevo. Este tipo de política de expiración es llamado expiración absoluta – una página es válida hasta cierto tiempo.

**¿Qué es connection pooling y cómo haces que una aplicación lo utilice?**

Abrir una conexión de base de datos es un operación que consume tiempo.

Connection pooling incrementa el rendimiento de las aplicaciones reutilizando las conexiones de base de datos activas en vez de crear una nueva conexión por cada petición.

El comportamiento del Connection pooling es controlado por los parámetros de la cadena de conexión.

Estos son los 4 parámetros que controlan la mayoria de los comportamientos del connection pooling.

* Connect Timeout
* Max Pool Size
* Min Pool Size
* Pooling

**¿Cuáles son los diferentes métodos de mantenimiento de sesión en ASP.NET?**

Hay 3 tipos:

* In-process storage
* Session State Service
* Microsoft SQL Server

**In-Process Storage**

Por defecto la sesión se almacena en el proceso de ASP.NET.

**Session State Service**

Como alternativa a usar el in-process storage para el estado de la sesión, ASP.NET provee el ASP.NET State Service. Te da un alternativa out-of-process para almacenar el estado de la sesión que no está amarrado tan de cerca al propio proceso de ASP.NET.

Para usar el State Service, necesitas editar el elemento sessionState en el web.config de tu aplicación ASP.NET.

También necesitarás iniciar el ASP.NET State Service en la computadora que has especificado en el atributo stateConnectionString. El .NET Framework instala este servicio pero por defecto se establece en inicio manual. Si vas a depender de él para almacenar el estado de la sesión, querrás cambiar el inicio a automático usando el plug-in del MMC (Microsoft Management Console) en el grupo “Herramientas Administrativas”.

Si haces esos cambios y luego repites los pasos previos, verás un ligero cambio en el comportamiento: el estado de la sesión persiste aun cuando reciclas el proceso de ASP.NET.

Hay 2 ventajas principales de usar el State Service. Primero, no corre en el mismo proceso de ASP.NET, si se cuelga el ASP.NET, eso no destruirá la información de sesión. Segundo, el stateConnectionString que es utilizado para ubicar el State Service incluye la dirección TCP/IP del servicio, el cual necesita estar corriendo en la misma computadora como ASP.NET. Esto permite que compartas información de estado en un web garden (múltiples procesadores en la misma computadora) o aún en un web farm (múltiples servidores corriendo la aplicación). Con el in-process storage por defecto no puedes compartir información de estado entre múltiples instancias de tu aplicación.

La mayor desventaja de usar el State Service es que es un proceso externo, en vez de ser parte de ASP.NET. Eso significa que leer y escribir información de sesión es más lento que lo que sería si mantuvieras el estado in-process.  
Y, de seguro, es un proceso más que necesita ser administrado. Como ejemplo del esfuerzo extra que esto trae, hay un error (bug) en la primera versión del State Service que permite a un atacante determinado hacer caer el proceso ASP.NET de manera remota. Si estás utilizando el State Service para almacenar el estado de la sesión, deberías instalar el parche del Microsoft Security Bulletin MS02-66 o instalar el Service Pack 2 del .NET Framework.

**Microsoft SQL Server.**

La opción final para almacenar la información de estado es salvarla en una base de datos de SQL Server. Para usar SQL Server para almacenar el estado de la sesión, necesitas ejecutar varios pasos de configuración.

Corre el script InstallSqlState.sql en el servidor SQL Server donde intentas almacenar el estado de la sesión. Este script creará los objetos de base de datos necesarios. El .NET Framework instalará este script en la misma carpeta donde están sus compiladores y otras herramientas – por ejemplo, C:\WINNT\Microsoft.NET\Framework\v1.0.3705 en una computadora Windows 2000 con la versión 1.0 del framework. Edita el elemento sessionState en el web.config de tu aplicación ASP.NET como sigue:

Indica el nombre del servidor, SQL Server permite compartir el estado de la sesión entre los procesadores de un web garden o los servidores de un web farm. Pero tienes el beneficio adicional del almacenamiento persistente. Aún si la computadora que tiene el SQL Server se cuelga y es reiniciada, la información de estado de la sesión aún estará presente en la base de datos y estará disponible tan pronto la base de datos esté ejecutándose de nuevo. Eso es porque SQL Server, siendo una base de datos de clase industrial, está diseñada para registrar sus operaciones y proteger tus datos a casi toda costa. Si estás dispuesto a invertir en SQL Server Clustering, puedes mantener el estado de la sesión disponible transparentemente para ASP.NET aún si el servidor primario de SQL Server se cuelga.

Como el State Service, SQL Server es más lento que mantener el estado de la sesión in-process. Necesitas poner adicionales pagos de licenciamiento para utilizar SQL Server para el estado de la sesión en una aplicación en producción. Y, de seguro, necesitas preocuparte por amenazas relativas a SQL Server tales como el gusano “Slammer”.

**¿Qué hace la propiedad “EnableViewState”? ¿Por qué quisiera prenderla o apagarla?**

EnableViewState activa la característica de administración automática de estado que permite a los controles de servidor re-poblar sus valores entre idas y venidas sin requerir que escribas ningún código. Esta característica no es gratis, desde que el estado de un control es pasado desde y hacia el servidor en un campo de formulario oculto. Deberías estar atento de cuando ViewState te está ayudando y cuando no. Por ejemplo, si estás enlazando un control a datos en cada ida y vuelta (como en el datagrid) entonces no necesitas que el control mantenga su ViewState, desde que tu lo limpiarás y re-poblarás los datos de todas formas. ViewState está activado para todos los controles de servidor por defecto. Para desactivarlo establece la propiedad EnableViewState a false.

**¿Cuál es la diferencia entre Server.Transfer y Response.Redirect?**

Server.Transfer se usa cuando queremos transferir la petición de la página actual a otra página .aspx en el mismo servidor, queremos preservar los recursos del servidor y evitar innecesarios viajes al servidor, queremos preservar el Query String y las variables del formulario (opcionalmente) y no necesitamos mostrar la URL real a donde la petición será redirigida en el navegador web del usuario. Para persistir información se requiere usar la colección Context.Item.

Response.Redirect se usa cuando queremos redirigir la petición a alguna página plana HTML o a algún otro servidor web, no nos importan causar varios viajes al servidor en cada petición, no necesitamos preservar el Query String y las variables del formulario de la petición original y queremos que nuestros usuarios puedan ver la nueva URL redirigida en su navegador (y que puedan agregarla a sus favoritos si fuera necesario).

1. En una aplicación de consola para leer los parámetros utilizaremos un **array** de variables de tipo **string** que llamaremos “**args**“.
2. Para cambiar el color de fondo de una aplicación de consola utilizaremos **Console. ColorBackground = System.Color(“Green”);**
3. La siguiente instrucción **Console.WriteLine(“\nChar: {0} y {1}”, Char.MinValue, Char.MaxValue);** nos devolverá una “a” minúscula y una “Z” mayúscula.
4. El tipo “**long**” es el tipo de número entero más alto que podemos representar en C#
5. El tipo “**decimal**” ocupa 128 bits (el doble que un tipo **long**) y es el tipo de número con decimales más grande que se puede representar en C#.
6. El tipo “**double**” ocupa lo mismo que el tipo “**ulong**“.
7. El tipo “**char**” ocupa 2 bytes que es lo mismo que el tipo entero “**short**” que a su vez también ocupa lo mismo que el tipo “**ushort**“, es decir 16 bits.
8. Las variables en C# a diferencia de Java, no distinguen entre mayúsculas y minúsculas.
9. No se pueden utilizar palabras reservadas como nombres de variables.
10. Las variables siempre empiezan por una letra o por el símbolo “**@**“.
11. Para escribir una comilla simple en una variable alfanumérica podemos utilizar la secuencia de escape \’
12. Este es un ejemplo de declaración y asignación de una variable lógica: **Bool EsCierto = true;**
13. Una variable tipo **Object** podemos almacenar cualquier cosa.
14. El tipo **dynamic** representa una dirección de memoria.
15. Mediante la instrucción **“$”** podemos asignar el valor **null** a cualquier variable.
16. Para saber si una variable tiene asignado algún valor distinto de **null** se puede utilizar la propiedad **HasValue**. Por ejemplo if (CodigoPostal.HasValue), siendo CodigoPostal de tipo “**int**” …
17. Para asignar el valor **null** a una variable tipo string podemos asignarla directamente **variable = null;** o bien asignado la cadena vacía **variable = “”**;
18. V1=null; V2=false; V3=(V1&V2)     ; V4=(V1|V2); V5=V3&V4; dará como resultado en V5 “**null**“.
19. La conversión explícita y = (long) x; daría un error de compilación si la variable x fuera de tipo double.
20. Console.WriteLine (**String.Format**(“{0:p}”,0.2)); dará como resultado **20,00%.**
21. Console.WriteLine (**String.Format** (“Importe = {0:e}”,12.35)); dará como resultado **12,35€.**
22. Console.WriteLine (**String.Format**(“{0:T}”,**DateTime.Now**)); dará como resultado la Hora del Sistema hasta los segundos.
23. El método **Parse** se utiliza para convertir una cadena en algún tipo numérico, de igual manera se utilizará el método **ToString** para pasar un tipo numérico a cadena.
24. El método **TryParse** es un método sobrecargado que devuelve un valor booleano true si la conversión se realiza correctamente y false en caso contrario.
25. En caso de no incializar una varibale numérica el valor inicial será **cero**; si es una cadena el carácter vacío, a valor **null** si es del tipo Object, y a true si es del tipo boolean.
26. Las inicializaciones por defecto de las variables no se produce cuando se declaran dentro de una función.
27. El ámbito de las variables puede ser a nivel de bloque, a nivel de función o a nivel de clase, siendo la de mayor ámbito la que se declara a nivel de bloque.
28. El nivel de acceso **public**no se pueden utilizar en el interior de las funciones.
29. El nivel de acceso **protected** restringe el acceso a la variable únicamente al código de la clase donde está definida, así como a las de todas las clases que hereden de esta.
30. El nivel de acceso **internal**no se puede utilizar en el interior de una función.
31. El nivel de acceso **private** no se puede utilizar en el interior de una función.
32. Sino indicamos nada a una variable; esta se considera como **public**.
33. No se puede calcular el valor de una constante a partir de una variable.
34. No se puede calcular el valor de una constante a partir de una constante en tiempo de ejecución, sí se puede en cambio en tiempo de compilación.
35. Todas las enumeraciones comienzan por el índice 0.
36. Las enumeraciones se pueden utilizar como un tipo de variable.
37. Todas las matrices de forma directa o indirecta hay que indicarles el tamaño que tienen sus dimensiones.
38. Para obtener el número de casillas de una matriz utilizaremos la función **GetLenght**
39. Para saber el tamaño de una dimensión de una matriz se utiliza la función **Lenght**(<número dimensión>)
40. Para saber las dimensiones de una matriz se utiliza la función **Ranking**
41. Es posible ordenar una matriz con la función **Array.Sort**
42. Siendo “cadena1” una variable de tipo **string** si queremos buscar la posición del primer carácter “**@**” de la cadena utilizaremos la instrucción **cadena1.IndexOf(“@”);** en caso de existir dos caracteres “**@**” la función devolverá el valor **-1** y en caso de no existir la función devolverá el valor **0**.
43. Para reemplazar en una cadena se puede utilizar el método **Replace.**
44. Las estructuras “**struct**” se construyen únicamente a partir de la combinación de otros tipos básicos del lenguaje C#.
45. Las estructuras “**struct**” pueden incluir funciones;
46. La clase **StringBuilder** permite entre otras funciones la concatenación de cadenas, es ligeramente más lenta que la concatenación directa con el operador “+” pero evita las excepciones en caso de concatenar tipos de datos **hetereogéneos**.
47. El operador “**Is**” devuelve un valor booleano indicando si la variable es del tipo que se indica.
48. La comparación **If (test1)&(test2)** es menos o igual de eficiente que **If (test1)&&(test2)**.
49. El operador “**%**” devuelve el cociente de una división entera.
50. La comparación **(test1)^(test2)** será cierta sólo si **test1** es **true** y **test2** es **true**.
51. Esta estructura con **if** sería equivalente a una estructura **switch**de 3 opciones y un default, **if** (condición) { … } **else**{ …**if** (condición) {…} **else** {…} }
52. En un **switch** si alguna opción no tiene la instrucción **break** el programa continuará evaluando las opciones inferiores del **switch**.
53. Una estructura **while (true) { Console.Write(“a”); }** provoca un error de compilación ya que detecta que estamos en un bucle infinito y no dejará compilar el programa.
54. Una estructura: **do** { … } **foreach** (<enumeración>); se ejecutará al menos una vez.
55. El siguiente bucle se ejecutará 8 veces **for (k1 = 23; k1 > 0; k1=(k1–)-3) { … }**
56. El siguiente código daría error de compilación porque se están comparando diferentes tipos de números: **double num\_pi = Math.PI; for (k1 = 23+(long) num\_pi; k1 > -2.56; k1=(k1–)-5/2) {…}**
57. El siguiente código intercambiaría el valor de las variables x e y sin utilizar una tercera **variable Int x=-10, y=+10; x = x + y + y; y = x – y – y;**
58. La visibilidad de un procedimiento de forma predeterminada es “**private**” a no ser que indiquemos otra opción en su declaración.
59. La ejecución de un **return** dentro de una función hace que la función asigne un valor a la instrucción llamante de la función y además termine la ejecución de dicha función aunque haya más cógido fuente por debajo de ella, de hecho el compilador nos dará una advertencia de que el código siguiente e **return** es inaccesible.
60. Los procedimientos de propiedades, también llamadas encapsuladores y utilizan las palabras reservadas get y set para acceder a la propiedad.
61. Existe la posibilidad de definir el comportamiento de un operador al usar como operando variables a priori no compatibles con el operador. Por ejemplo: public static Cliente **operator**  
    **+** (Cliente cl1, Cliente cl2) {… return …}
62. Las variables por referencia siempre deben ir precedidas de la palabra reservada “**ref**” o “**out**“.
63. Las palabras reservadas “**ref**” y “**out**” deben ser utilizadas tanto en la llamada a la función como en la declaración de parámetros del procedimiento o función.
64. La diferencia entre “**ref**” y “**out**” es que mientras las variables que se pasan por “**ref**” no necesitan ser inicializadas, las variables que se pasan por “**out**” necesitan obligatoriamente ser pasadas inicializadas.
65. Cuando creamos un procedimiento o función que no está sobrecargado ni tiene parámetros opcionales, debemos saber a priori el número exacto de parámetros que le vamos a pasar, es decir el número de parámetros está determinado por la propia declaración de la función o procedimiento.
66. Es posible crear parámetros opcionales, asignándole un valor por defecto en la declaración, además hay que tener la precaución de inicializar a partir del primer parámetro inicializado, el resto de parámetros de la derecha para que ocurra ningún error de compilación.
67. Cuando se hace la llamada a una función o procedimiento con parámetros opcionales habría que dejar los huecos en la llamada para que el compilador sepa a qué parámetro nos estamos refiriendo, por ejemplo en una llamada a una función con 3 parámetros donde el primero es obligatorio y los dos últimos opcionales una posible llamada a esa función sería **resultado = FuncioLoQue Sea (25, , ref variable3);**
68. Los parámetros siempre deben llevar el orden en que fueron declarados excepto el caso en que sean nominados, en cuyo caso siempre podemos ponerlos como queramos, por ejemplo **FuncionLoQueSea(25, parametro2: 33, “€”);** siendo los parámetros del a la función Int parametro1,Int parametro2 y String parametro3.
69. Esta declaración de función **public**  
    **double Media (int[] notas)** está mal le faltaría la palabra reservada **params** ya que estamos pasando como parámetro un array.
70. Esta declaración de función **public**  
    **double Media (int[] notas)** está bien si lo que le pasamos a la función es una variable por referencia que apunte a un array de enteros.
71. Esta declaración de función **public**  
    **static**  
    **double Media (param int[] notas)** está bien si le pasamos por valor una serie cualquiera de números enteros, por ejemplo: **Media(4\*2, 2+3, 7-7)**.
72. Los **objetos** son la representación simbólica de las **clases**.
73. Las tres principales propiedades de los objetos son: **la herencia, el polimorfismo y la instanciación**.
74. El **UML** es una clase abstracta propia del lenguaje C#.
75. Tenemos 7 posibles combinaciones para determinar la visibilidad de una clase**: public, internal, private, private internal, protected, abstract, sealed.**
76. **Protected**: no permite crear instancias de esta clase, sólo sirve para ser heredada como clase base. Suelen tener los métodos definidos pero sin ninguna operatividad con lo que se suele escribir estos métodos en las clases derivadas.
77. **Internal**: cuando una clase es la última de una jerarquía, por lo que no podrá ser utilizada como base de otra clase.
78. **Private internal:**la clase tendrá las mismas propiedades que private e internal.
79. **Private:** la clase sólo puede usarse en el módulo en la que está definida.
80. Cuando declaramos una clase como parcial (**partial**) le indicamos al compilador que vamos a definir una clase en diferentes declaraciones, incluso con la posibilidad de reescribir algún método de la clase en una segunda o tercera declaración parcial de la clase.
81. Cuando declaramos alguna propiedad como **private** la única forma de acceder a ella será a través de los métodos o de la encapsulación utilizando las funciones “**get**” y “**set**“.
82. Un ejemplo de encapsulación sería: public **String Apellido { get{ elApellido = value.ToUpper(); } set {return elApellido;}}**donde elApellido sería la propiedad **private**.
83. Se pueden restringir los accesos a una propiedad, si sólo incluimos la opción “**get**” lo que estamos haciendo es dando permisos de lectura.
84. Si incluimos la opción “**set**” lo que hacemos es dar permisos de lectura y escritura.
85. No es posible crear una propiedad que haga referencia (es decir sea del mismo tipo) a la propia clase que estamos creando, ya que produciría un error de compilación. Es decir: **public class Persona {… private Persona Pepe = new Persona(); … }** daría error al hacer una referencia circular.
86. Si es posible crear una propiedad que sea un array de objetos de la misma clase que la clase que estamos creando . Es decir: **public class Persona {… private Persona[] losHijos = new Persona[10]; ….}**
87. Todos los métodos de una clase pueden manejar campos de la clase incluso si son privados.
88. La **sobrecarga** es la creación dentro de la clase, de un grupo de métodos que tienen el mismo nombre pero con un número de parámetros distinto aunque los parámetros coincidentes tienen que ser del mismo tipos de datos.
89. Las clases derivadas siempre heredan las propiedades y métodos de su clase base.
90. Las propiedades privadas no son heredadas por las clases hijas, simplemente heredarán los métodos que acceden a esas propiedades privadas.
91. Para poder sobrescribir un método en una clase hija, se utilizará la palabra reservada **over**en la clase hija y **virtual**en la clase heredada.
92. El siguiente método que se sobrescribe: **public sealed void visualización () { base.visualizacion();}** haría lo mismo que el método heredado, pero con la diferencia de que ese método ya no podría reescribirse en otras clases que lo hereden ya que incluye la directiva **sealed**.
93. Cuando declaramos en una clase un método como abstracto (**abstract**) estamos indicándole al compilador que el método no tiene implementación sólo definición.
94. Cuando un método es marcado como abstracto entonces la clase a la que pertenece también tiene que ser marcada como abstracto (**abstract**).
95. Además de poder hacer clases parciales (**partial**) también es posible hacer métodos parciales.
96. Para poder acceder a las propiedades de un objeto utilizaremos la siguiente sintaxis: **<nombre del objeto>.<propiedad>**.
97. Si estamos realizando una aplicación en Windows Form el objeto más aconsejable para la introducción de datos por parte del usuario es el objeto tipo **Label**
98. Un **ListBox** nos muestra un campo donde sólo se visualiza un registro pero puede desplegarse para ver otros valores.
99. Utilizaremos un **TextBlock** para mostrar el contenido de texto de varias líneas.
100. Uno de los métodos del objeto **Button** es el **OnClickMouse**

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1 V** | **2 F** | **3 F** | **4 F** | **5 V** | **6 V** | **7 V** | **8 F** | **9 V** | **10 V** |
| **11 V** | **12 F** | **13 V** | **14 F** | **15 F** | **16 V** | **17 F** | **18 F** | **19 F** | **20 V** |
| **21 F** | **22 V** | **23 V** | **24 V** | **25 F** | **26 V** | **27 F** | **28 V** | **29 V** | **30 V** |
| **31 V** | **32 F** | **33 V** | **34 V** | **35 F** | **36 V** | **37 V** | **38 F** | **39 F** | **40 F** |
| **41 V** | **42 F** | **43 V** | **44 F** | **45 V** | **46 F** | **47 V** | **48 V** | **49 F** | **50 F** |
| **51 F** | **52 F** | **53 F** | **54 F** | **55 V** | **56 F** | **57 V** | **58 F** | **59 V** | **60 V** |
| **61 V** | **62 F** | **63 V** | **64 F** | **65 F** | **66 V** | **67 F** | **68 F** | **69 F** | **70 V** |
| **71 V** | **72 F** | **73 F** | **74 F** | **75 F** | **76 F** | **77 F** | **78 F** | **79 V** | **80 F** |
| **81 V** | **82 F** | **83 V** | **84 F** | **85 F** | **86 V** | **87 V** | **88 F** | **89 V** | **90 F** |
| **91 F** | **92 F** | **93 V** | **94 V** | **95 V** | **96 V** | **97 F** | **98 F** | **99 V** | **100 F** |

**Carlos**:

**1. Cuántos bits ocupan los siguientes tipos numéricos:**

1. byte: 4 bits, long: 32 bits, short: 16 bits y decimal: 132 bits.
2. byte: 8 bits, long: 64 bits, short: 16 bits y decimal: 128 bits.
3. byte: 4 bits, long: 32 bits, short: 24 bits y decimal: 128 bits.
4. byte: 8 bits, long: 64 bits, short: 16 bits y decimal: 132 bits.

**Respuesta: b)**

**2. Según el código:**

class Test

{

staticvoid Main()

{

double x = 1243.73;

int c;

int a;

a = (int) x;

    c = 1234.73;

System.Console.WriteLine(a);

}

}  
  
**Que salida se obtendría y de que tipo?**

1. “1243.73” de tipo int.
2. “1243” de tipo int.
3. “1234” de tipo double.
4. “1243.73” de tipo double.

**Respuesta: b)**

**3. Qué significa este ejemplo “If (test1)&&(test2)” en relación al operador “&&”?**

1. Solo evalua test2 si test1 es cierto.
2. Cierto si ambos son ciertos.
3. Cierto si alguno de los dos es cierto.
4. Ninguna respuesta es correcta.

**Respuesta: a)**

**Tomás**

1.-En caso de que no se le de valor a un tipo bolean el valor por defecto es ? :  
a)True.  
b)False.  
c)Null.  
d)Hay que darle un valor obligatoriamente.  
e)Todas son falsas.

La solución es : b)False.

2.-Se le llama \_\_\_\_\_\_\_\_\_\_\_\_\_ al formato monetario tal como está definido en el sistema.   
String.format (“Importe = {0:c}”,12.35);   
Resultado: 12,35 €   
La solución es :Currency

3.-Cuanto es un byte?  
a)16 bits  
b)9 bits  
c)16 kb  
d)9 kb  
e) a y c son correctas  
f) b y d son correctas  
g) Ninguna es correcta

La solución es g) Ninguna es correcta

**Ignacio**:

2.- ¿Como podemos hacer un cambio de variable de tipo double a string?

* 1. Double y = 55;

String x = y.ToString();

* 1. Double x = 55;

String y = Double.x.ToString();

* 1. Double x;

x = Double.parseDouble(textBox.Text);

* 1. Ninguna es correcta.

RESPUESTA CORRECTA: 1

3.- ¿Cual de las siguientes afirmaciones es correcta?

* 1. La palabra this hace referencia a la instancia actual de la clase.
  2. Se utiliza como modificador del primer parámetro de un método de extensión.
  3. Se utiliza para obtener acceso a miembros con el fin de evitar ambigüedades con nombres similares.
  4. Todas son correctas.

RESPUESTA CORRECTA : 4

**Álvaro**

La forma del resultado de la conversion String.format(“{0:f}”,0.2); es:

a) Resultado: 20,00%  
b) Resultado: 2,458136e+005  
c) Resultado: 0,20  
d) Ninga es cierta  
Respuesta c

Crear un OBJETO a partir de un una clase es lo que se llama \_\_\_\_\_\_\_\_\_.  
   
a) INSTANCIAR  
b) ENCAPSULACIÓN  
c) CLASES DERIVADAS  
d) Todas son cierta  
Respuesta a  
   
   
Las tres propiedades de la ORIENTACIÓN A OBJETOS son:

a) Private, set, get  
b) String, int, double  
c) Encapsulacion, herencia, polimorfismo  
d) Ninguna es cierta  
Respuesta c

**Alvaro Laguna:**

**1) Un Objeto es:**

1. Una entidad con características similares a otro objeto

2. Una entidad con características diferentes a otro objeto

3. Una entidad con características propias

4. Ninguna de las anteriores

**Respuestas: 3**

**2) Que objeto utilizaremos para receptar un dato detro de un formulario**

1. Button

2. Radiobutton

3. Label

4. Textbox

**Respuesta: 4**

**3) La aplicación Windows forms me permite**

1. Crear un formulario en blanco para agregar objetos

2. Crear un contenedor de objetos

3. Crear una forma con varios objetos

4. Ninguna de las anteriores

**Respuesta: 1**

**Daniel López**

**1. ¿ Cuál de ellos no es una estructura de control?**

a) if

b) try

c) while

d) switch

**Respuesta: b**

**2. ¿Que mostrará el siguiente código?**

class Ejercicio {

static void Main(String[] args) {

int var1 = 4;

String var2 = “La variable 1 vale “;

if (var1 < 4) {

int var3 = 2;

int var4 = var1 + var3;

Console.WriteLine(“La varible 4 vale: ” + var4);

}

Console.WriteLine(var2 + var1);

a) Se produce error

b) La variable 4 vale: 6

c) La variable 4 vale: 6 La variable 1 vale: 4

d) La variable 1 vale: 4

**Respuesta: d**

**3. ¿Si ponemos a una variable un nombre reservado, que cáracter debemos poner para que no lo sea?**

a) /

b) ?

c) @

d) %

**Respuesta: c**

**Rubén**

1-Los tipos de variables enteras :  
a)char , String  
b) byte ,  ushort , uint , ulong    
c) float ,double , decimal.   
d) b y c son correctas

Correcta : respuesta b)

2-En este fragmento de codigo, cual es la salida por pantalla:

class Program  
{  
    static void Main(String[]args)  
    {

        int x=5, y=5;  
          
        for (int i=0;  i<y;  i–){  
        Console.write(i+”,”);  
    }

}}

a) imprimira 0,2,3,4  
b)imprimira 0,-1,-2,-3,-4 y se parara  
c)sera un bucle infinito que empiece con el 0 y siga decrementando infinitamente   
d) ninguna es correcta

Correcta: respuesta c).

3- Elige la salida correcta por pantalla

class Program  
{  
    static void Main(String[]args)  
    {

    int [] array = new int [10];  
    int [] vector = new int[5];  
        for (int i=0; i<array.Length ; i++) {  
          
        array[i]=99;  
}

Console.WriteLine(array[0]);  
    }

}}

a)0  
b)10  
c)99  
d)5

Correcta: respuesta c).

4-elige la correcta:

a) \n= salto de pagina   
b) \n= salto de línea   
c) \n= tabulacion horizontal  
d) \n= retorno de carro

Correcta: respuesta b).

**Jonathan**

**1- ¿Cuales son las propiedades de la orientacion a objetos?**

a) Encapsulación

b) Herencia

c) Polimorfismo

d) Todas son ciertas

**Respuesta: d**

**2- ¿Que secuencia de escape utilizaremos para un salto de linea?**

a) \n

b) \’

c) \\

d) \b

**Respuesta: a**

**3- ¿Que visualizará el siguiente código?**

**string** cadena =  
“123”;

Console.writeLine  
(cadena +  
456);

a) 123  
b) 456  
c) 123456  
d) 579

**Respuesta: c**

**Geordano**

* 1. **TIPOS DE VARIABLES NUMÉRICAS DECIMALES:**  
     a). Short – Float – Long  
     b) Double – Float – Int  
     ***c)*** Double – Float – Decimal  
     d) Ningunas de las anteriores
  2. **Cual de las siguientes, es una definición correcta de un array de objetos:**  
     ***a)***private Persona[] p = new Persona[10];  
     b)private persona p [10];  
     c)Persona[] = new Persona ;  
     d)Todas son correctas
  3. **Cual sería la salida del siguiente código:**
  4. double x;
  5. long y;
  6. x = 21.234323;
  7. y = (long) x;

Console.WriteLine (“valor de y:” + y);

a)valor de y: 21.234323  
b)valor de y: 21.23  
***c)***valor de y: 21

**Daniel**

**1) La visibilidad de un procedimiento viene determinada por la declaración:**

1. Private, public o internal.

 2. Return.

 3. Sólo public.

 4. Todas son falsas.

**Respuesta correcta: 1.**

**2) La ejecución de***return***provoca:**

 1. El inicio del proceso.

 2. La salida de la función.

 3. Una nueva sentencia.

 4. Todas son ciertas.

**Respuesta correcta: 2.**

**3) La variable instancia:**

 1. Es el objeto de la clase.

 2. Representa un atributo de un objeto.

 3. Es el método de una clase.

 4. Ninguna de las anteriores es cierta.

**Respuesta correcta: 2.**

**Eugenia**

1) Teniendo esta instrucción:

void CambiarValor() {  
  Console.WriteLine(“¡¡¡Aprobado!!!”);  
}

Qué procedimiento visible, por defecto se indica:  
1.-Private.  
2.-Public  
3.-Protected  
4.-Ninguno  
**Respuesta: 2.-Public**

2) Da error este código al compilar?:  
int a = 5;               
int b = a + 2;   
bool test = true;  
int c = a + test;

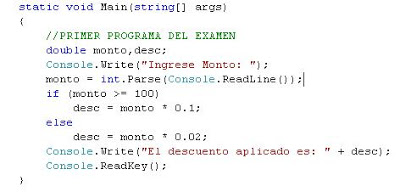
1.-No, no hay error y compilará  
2.-Hay  error en la segunda línea.  
3.-Hay  error en la tercera línea.  
4.-Hay error en la cuarta línea.  
**Respuesta: 4.-Hay error en la cuarta línea.**

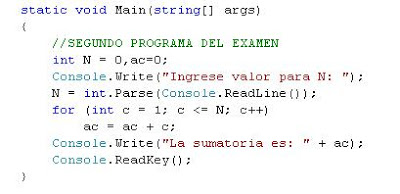
3) ¿Que aparecerá al ejecutar este código?  
Double myDouble = 42.72;  
 int myInt = Convert.ToInt32(myDouble);  
   Console.WriteLine(myInt);

1.-42.72  
2.-42  
3.-No se ejecutará, producirá error  
4.-43  
**Respuesta: 4.- 43**

### **Resolucion Examen C#**

1. Dado un monto ingresado desde teclado. Calcular el descuento considerando que si el monto está por encima de 100, el descuento es del 10%, y por debajo de 100, el descuento es del orden del 2%.

2. Calcular mediante un algoritmo repetitivo la suma de los primeros N numeros Naturales. Siendo N un dato ingresado desde teclado.

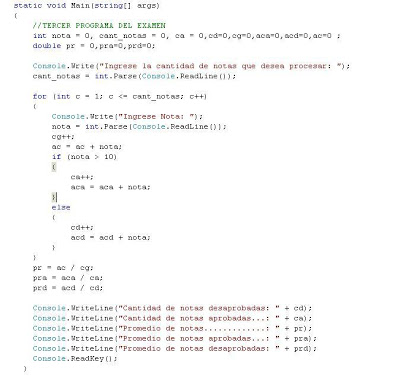
3. Dadas N notas de un estudiante, calcular:

a) Cantidad de notas desaprobadas

b) Cantidad de notas aprobadas

c) Promedio de notas

d) Promedio de notas aprobadas y Promedio de notas desaprobadas



**4.**

¿Qué devuelve la siguiente función? ¿por qué) ? using System;class Q1: System.NullReferenceException {  static int Main() {    try {      string valor = "Value: " + (Q1)null;      return valor.Length;    }    catch (Q1 q) {      throw;    }  }  public static string operator +(string msg, Q1 q) {    return msg + q;  }} Posibles respuestas

* Retorna la longitud de la variable valor
* Se produce un NullReferenceException
* Se produce un StackOverflowException

¿Cuál es la diferencia entre las excepciones producidas por Foo1 y Foo2? using System;class Program{    static void Main(string[] args)    {        try {            Foo1();        }        catch (Exception ex) {            Console.WriteLine(ex.ToString());        }        Console.WriteLine("------------------");        try {            Foo2();        }        catch (Exception ex) {            Console.WriteLine(ex.ToString());        }    }    private static void Foo1()    {        try {            Bar();        }        catch (Exception ex) {            throw ex;        }    }    private static void Foo2()    {        try {            Bar();        }        catch (Exception ex) {            throw;        }    }    private static void Bar()    {        throw new Exception("Error!");    }}

Un Evento es...

* A.

El resultado de la acción de un usuario.

* B.

El código que fuerza una acción de usuario

Propiedades, métodos y eventos de los controles más importantes 1 Introducción. Propiedades comunes T 2 El formulario H 3 Control etiqueta T 4 Control cuadro de texto H 5 Control Botón V 6 Practica tú mismo P 7 Control cuadro combinado H 8 Control lista H 9 Control imagen H 10 Control temporizador H 11 Control menú V 12 Practica tú mismo P 13 Otros controles útiles T 14 Cuestionario C 10 Acceso a datos con ADO.NET desde C# 1 Introducción H 2 Acceso a SQL Express 2005 desde C V 3 Conceptos básicos sobre el diseño de una base de datos V 4 Practica tú mismo P 5 Objeto DataSet V 6 Acceso a los datos desde la interfaz: Control DataGridView V 7 Manipulación de datos: Aviso importante H 8 Eliminación de registros con DataGridView V 9 Inserción de registros con DataGridView V 10 Acceso a los datos con otros controles visuales V 11 Practica tú mismo P 12 Cuestionario C 11 Anexo: tareas más habituales con Visual Studio 2005 1 Obtención de ayuda V 2 Depuración de aplicaciones V 3 Creación de un programa de instalación para la aplicación

Fundamentos de C#: Programación orientada a objetos 1 Clases y objetos: Introducción H 2 Declaración de una clase V 3 Propiedades o atributos de una clase H 4 Métodos de una clase H 5 Verdadero o falso vf 6 Utilización de las clases. Clases instanciadas y no H 7 Practica tú mismo P 8 Constructores y destructores H 9 Herencia H 10 Polimorfismo H 11 Practica tú mismo P 12 Cuestionario C 7 Clases, métodos y propiedades útiles de C# 1 Introducción H 2 Funciones para el manejo de arrays H 3 Practica tú mismo P 4 Funciones para el manejo de cadenas H 5 Practica tú mismo P 6 Funciones para la conversión de tipos V 7 Practica tú mismo P 8 Cuestionario C 8 Introducción al diseño de una interfaz de usuario en C# 1 Introducción H 2 El entorno de trabajo T 3 Controles. Dibujar los controles en el formulario V 4 Manipulación de los controles visuales V 5 Propiedades de los controles V 6 Practica tú mismo P 7 Definir los controladores de eventos T 8 Manipulación de los controles desde código T 9 Practica tú mismo P 10 Cuestionario

Fundamentos de C#: Tipos de datos y trabajo con variables 1 Tipos de datos básicos V 2 Las variables en C V 3 Declaración de una variable H 4 Alcance de las variables T 5 Relaciona conceptos rc 6 Literales H 7 Operadores y expresiones H 8 Constantes V 9 Verdadero o falso vf 10 Arrays: Introducción H 11 Trabajando con arrays T 12 Practica tú mismo P 13 Cuestionario C 4 Fundamentos de C#: Estructuras de control 1 Sentencias de control: ¿Para qué sirven? V 2 Condiciones. Operaciones de comparación H 3 Condiciones complejas. Operadores booleanos H 4 Verdadero o falso vf 5 Sentencia If V 6 Sentencias If más complejas V 7 Sentencia switch V 8 Practica tú mismo P 9 Cuestionario C 5 Fundamentos de C#: Estructuras de repetición o bucles 1 Bucles: ¿Para qué sirven? V 2 Sentencia for H 3 Sentencia for en acción V 4 Practica tú mismo P 5 Sentencia foreach H 6 Colecciones de datos y uso de foreach H 7 Sentencia while H 8 Practica tú mismo P 9 Cuestionario

1 .NET Framework T 2 Common Language Runtime (CLR) T 3 Microsoft Intermediate Language (MSIL) T 4 Veradero o falso vf 5 NameSpaces T 6 En su sitio st 7 Descripción del entorno de desarrollo IDE V 8 Explorador de soluciones T 9 Panel de propiedades V 10 Otros paneles del IDE Visual Studio 2005 V 11 Relaciona conceptos rc 12 Cuestionario C 2 Desarrollo del primer programa 1 ¿Qué es un proyecto? V 2 Tipos de proyectos V 3 Crear un nuevo proyecto V 4 En su sitio st 5 El entorno de trabajo T 6 Un vistazo rápido al código generado automáticamente T 7 Algunos aspectos importantes del lenguaje V 8 Escribiendo nuestra primera sentencia V 9 Ejecución del programa V 10 Practica tú mismo P 11 Cuestionario

## Pregunta 1

static void Test(out int x, out int y)

{

x = 42;

y = 123;

Console.WriteLine(x == y);

}

1. Falso
2. Cierto
3. Puede ser cierto
4. No compila

### **Solución – Pedro González (**[**@pedrogovi**](https://twitter.com/pedrogovi)**)**

La palabra reservada **[out](https://msdn.microsoft.com/es-es/library/t3c3bfhx.aspx)** hace que los parámetros del ejemplo sean pasados por referencia. El efecto de pasar parámetros por referencia es que cualquier cambio de estos parámetros dentro del método se refleje fuera del mismo. En este caso, la palabra reservada **out** tiene el mismo efecto que **ref**, salvo que esta última requiere que la variable sea inicializada previamente.

En el código que aquí se muestra se están cambiando los valores que tienen la variables que se ha pasado por referencia: x e y.

Si suponemos que la variable que se ha pasado por referencia como parámetro x no es la misma que la variable que se ha pasado como parámetro y, podríamos afirmar que el resultado es quexe y tendrán valores diferentes. Véase el siguiente ejemplo donde se realiza una llamada a dicho método con variables diferentes.

static void Main()

{

int a = 1;

int b = 2;

Test(out a, out b);

}

Sin embargo, si la llamada al método Test se realiza con la misma variable, como se muestra en el ejemplo siguiente, se estaría cambiando el valor de la misma variable y por lo tanto los valores de x e y serían iguales (x = 123, y = 123). No sabemos si x e y van a tener el mismo valor, pero podemos afirmar que podrían tenerlo y la respuesta correcta sería la 3.

static void Main()

{

int a = 1;

Test(out a, out a);

}

Para más información podéis visitar la [documentación oficial](https://msdn.microsoft.com/es-es/library/t3c3bfhx.aspx).

**Solución**: 3. Puede ser cierto

## Pregunta 2

static void Main(string[] args)

{

float value = 100000000;

while(value > 0)

{

–value;

}

Console.WriteLine(“Value es {0}”, value);

}

1. Value es 0
2. Value es 0.00000001
3. Nunca se imprime el valor de Value
4. No compila

### **Solución – Francisco Olmedo (**[**@fmolmedo**](https://twitter.com/fmolmedo)**)**

Repasamos los posibles puntos de interés:

En primer lugar, la asignación de la variable valuees correcta, ya que se hace conversión implícita de entero a float.

Posteriormente, la sentencia –value equivaldría a value = value – 1, por lo que también es correcta.

Por lo tanto, nos queda incidir en la pérdida de precisión por usar una variable de tipo float. Para su representación se utilizan 32 bits, de los cuales 1 es para el signo, 23 para la representación de la mantisa y 8 para el exponente. Por lo tanto, si tenemos 23 bits (24 si consideramos que el bit más significativo se asume a 1) para la mantisa, podemos representar: log10(2^24) ~= 7 dígitos significativos. Por lo tanto, el rango de enteros que se pueden representar con un float sin pérdida de precisión es el siguiente: [-16777216 – 16777216]. El número que le asignamos no entra en ese rango, con lo que, la operación de restar uno no se llevaría a cabo como se espera; y, consecuentemente, no se llegaría nunca a salir del bucle.

**Solución**: 3. Nunca se imprime el valor de Value

## Pregunta 3

int x = 1;

x = x++;

1. x==1
2. x==2
3. No existe el operador ++
4. No compila

### **Solución – Sergio Gallardo (**[**@maktub82**](https://twitter.com/maktub82)**)**

El [operador ++](https://msdn.microsoft.com/es-es/library/36x43w8w.aspx) es un operador de increment y puede aparecen antes o después del operando.

Cuando aparece antes del operando (++x) el resultado de la operación es el del operando después de incrementarse. Pero si aparece después del operando (x++), como en nuestro caso, el resultado de la operación es el del operando antes de haberse incrementado.

Por tanto la asignación se hace con el valor original de x, que es 1. Así que la respuesta correcta es **x==1**.

Sin embargo, si el operador ++ apareciera delante del operando (x = ++x), el resultado de la asignación sería el del operando después de haber sido incrementado y por tanto x valdría 2.

**Solución:**1. x==1

## Pregunta 4

static void Main(string[] args)

{

double result1 = 3.65d + 0.05d;

float result2 = 3.65f + 0.05f;

Console.WriteLine(result1 == 3.7d);

Console.WriteLine(result2 == 3.7f);

Console.WriteLine(result1 == result2);

}

1. TRUE, TRUE, TRUE
2. TRUE, FALSE, TRUE
3. FALSE, FALSE, FALSE
4. TRUE, TRUE, FALSE
5. FALSE, TRUE, TRUE
6. FALSE, TRUE, FALSE

### **Solución – Daniel Martín (**[**@danimart1991**](https://twitter.com/danimart1991)**)**

Cuando un ordenador necesita contener datos puede resultar sencillo, por ejemplo, un entero, con su representación estandarizada binaria, o puede resultar muy complicado, por ejemplo, un número decimal. Puede parecer sencillo, divides en dos partes el número, que pueden ser representados por dos enteros, y ambos enteros pueden ser transformados a su representación binaria y guardados en el sistema, pero este concepto puede dar problemas, por ejemplo, al intentar guardar un número irracional, o algo más sencillo, un tercio, al obtener el valor en coma flotante de un tercio, se obtiene un número periódico 0,333…, este valor no puede ser representado de manera sencilla.

Una solución que se propone en muchos sistemas es la aproximación. Esta aproximación, pasa por dar el número próximo más cercano disponible, pero este punto puede dar problemas de cálculo, inclusive en su transformación a binario. Por ejemplo, y tal y como ocurre en la pregunta 4: 3.65d + 0.05d no siempre puede resultar en 3.7d, y dependerá del lenguaje, el compilador y de cómo se realice la acción.

En C# existen varios tipos de números de coma flotante:

–          Float: un número de coma flotante de tipo 32 bit (1 bit para signo, 23 bits para mantisa, y 8 bits de exponente).

–          Double: un número de coma flotante de tipo 64 bit (1 bit para signo, 52 bits de mantisa, y 11 bits de exponente).

En el código de la pregunta al guardar el resultado double realiza un redondeo, y guarda el resultado más próximo disponible que es 3.6999999999999997. Al guardar el resultado float, si realiza un redondeo más acorde a lo esperado y si guarda correctamente el valor 3.7, por tanto, al comprarlo, es cierto.

[Más información](http://www.extremeoptimization.com/resources/Articles/FPDotNetConceptsAndFormats.aspx) sobre los número es coma flotante en C#

**Solución:** 6. FALSE, TRUE, FALSE

## Pregunta 5

static void Main(string[] args)

{

double delta = 0.1;

int i = 0;

double value = 0d;

for (; i < 100; i++)

{

value += delta;

}

Console.WriteLine(value);

Console.ReadLine();

}

1. Evidentemente no compila
2. 10
3. Casi 10
4. No termina
5. 1

### **Solución – Daniel Martín (**[**@danimart1991**](https://twitter.com/danimart1991)**)**

En esta pregunta ocurre lo mismo cuando se guarda el resultado. Conforme se incrementa value se va redondeando, hasta que al final, casi llega a 10. Debido al redondeo del double se queda en 9,99999999999998.

**Solución:** 3. Casi 10.