Tecnología y paradigmas de programación. Laboratorio 5. Grupos 1, 2 y 3.

(**NOTA**: eliminar cualquier referencia a LINQ de los fuentes. Salvo indicación en sentido contrario, no se usará ningún método de ninguna colección.)

1. Implementar las funciones trigonométricas hiperbólicas inversas usando delegados y notación Lambda (de las dos formas, con distintos nombres obviamente). Estas son las definiciones de estas funciones:
   asinh(x)=log(x+sqrt(x^2+1))
   acosh(x)=log(x+sqrt(x^2-1))
   atanh(x)=(log(1+x)-log(1-x))/2
   (En realidad podríamos hacer un ejercicio bonito de aplicación de las técnicas vistas en sesiones anteriores lanzando y tratando las excepciones adecuadas para los valores inválidos de x (ver artículo de la wikipedia). Queda como ejercicio propuesto.)
   Invocar esas funciones, mostrar el resultado por pantalla.

2. Implementar `static void AplicaAccion<T>(IEnumerable<T> e,Action<T> a)`, que aplica la acción `a` a cada elemento de `e`. Para ello usar `foreach`. Generar un array de enteros con valor aleatorio. Usar `AplicaAccion` para mostrar por la pantalla sus elementos en una misma línea, separados por espacios. ¿Cómo es la acción que hay que pasarle?

3. Impementar `static int Contar<T>(T[] v, Predicate<T> p)` recibe un vector de T y un predicado, devuelve cuantos elementos de v cumplen p.
   1. Contar cuantos pares hay en el array usando el predicado `isEven` del código de ejemplo (proyecto `predefined`) y `Contar`.
   2. Escribir un predicado que devuelva `true` si el entero que se le pasa es primo, `false` en caso contrario. Usarlo para mostrar cuantos primos hay en el vector aleatorio generado.
      Ejercicio opcional: reescribir los apartados anteriores definiendo el predicado en la llamada a `Contar`.

4. Implementar `static int PosicionPrimero<T>(T[] v, Predicate<T> p)` recibe un vector de T y un predicado, devuelve la posición del primer elemento de `v` que cumple `p`, -1 si no existe. Usar para mostrar la posición del primer par y del primer primo, usando los predicados del ejercicio anterior.

5. Partiendo de la función de orden superior `DoubleApplication` del código de ejemplo como modelo, escribir una función de orden superior que realice la composición de dos funciones que reciben y devuelven `double`. El nombre puede ser `ComposeDouble`. La composición de f con g, denotada (f ○ g) es g(f). La invocación de (f ○ g) (x) es igual a g(f(x)). Téngase en cuenta que la función de orden superior devuelve `Func<double,double>` y recibe dos `Func<double,double>`. Asimismo, como el resultado de la invocación es una función que recibe un `double` y devuelve otro `double`, puede aplicarse a un `double` y el resultado mostrarse por la pantalla. Probar la implementación comprobando que el logaritmo neperiano de la exponencial es la función identidad: log(exp(x))=x, es decir, log(exp(3.0)) es 3.0.

   Ejercicio opcional retorcido:
   Repetir esto mismo en una sola línea (lógica), la definición de `ComposeDouble` (anónima), la invocación con la exponencial y el logaritmo natural y la invocación del resultado pasandole un `double`, todo usando notación Lambda.

6. Escribir la función de orden superior genérica `ComposeG` que compone f con g en estas condiciones: f recibe T1 y devuelve T2, g recibe T2 y devuelve T3.¿Qué devuelve (f ○ g)? Intentar reescribir la composición de log y exp usando esta función, es delicada la inferencia de tipos en este caso.

   Ejercicio opcional alambicado:
   Usando `ComposeG` dos veces (anidadas), definiendo los delegados `Incrementa` (incrementa un entero), `ToInt` (convierte `char` en `int`), `ToChar` (convierte `char` en `int`), obtener `SiguienteChar,` que recibe un char y devuelve el siguiente, por ejemplo `SiguienteChar('a')` devuelve `'b'`. Probar la invocación.

7. Obtener la lista de los primos menores que un entero dado, usando la siguiente estrategia:
   Inicializar la lista de primos con el valor 2, el primer primo. Para cada número menor que el máximo, empezando por el siguiente al primer primo, comprobar que no existe un primo en la lista que sea su divisor. Si efectivamente no existe, añadirlo a la lista de primos. Para ello implementar
   `static bool Existe(List<int> v, Predicate<int> f)`
   devuelve true si existe al menos un `i` tal que `f(v[i])` es `true`, `false` en caso contrario.
   Este método es el que se usa para comprobar si existe un divisor del número que estamos comprobando si es o no primo en la lista de primos obtenidos. ¿Cuál será la `f` que hay que pasar a `Existe`?

NOTA: recordar entregar la tarea autónoma antes del siguiente laboratorio, en el examen de prácticas se usará.