4. Objetos

Los programas crecen...

- Los *arrays* y los *structs* nos permiten representar y agrupar los datos de manera estructurada (estructuras de datos).
- Después, los métodos pueden acceder a esas estructuras para gestionar los datos (añadir, modificar, eliminar, ...)

Pero nuestros programas crecen...

- ¿Cómo planteamos el diseño de un programa grande? ¿Cómo organizamos el código para trabajar con equipo de programadores?
  - ¿Que piezas diseñamos?
  - ¿Cómo facilitamos la modularidad (poder reemplazar código por otro distinto, pero con el mismo cometido)?
  - ¿Qué abstracción común planteamos para poder ensamblar las piezas?
  - ¿Cómo facilitamos la reutilización de código (no reimplementar las mismas funcionalidades)?

$\rightsquigarrow$ Programación orientada a objetos

Revisitando. . .

Algunas ideas sueltas para reflexionar . . .

- Hemos visto cómo implementar los polinomios
  - una representación para ellos
  - un conjunto de operaciones sobre ellos

- Si pudiésemos encapsular en un *paquete*

  $$\boxed{\text{tipo de datos}} + \boxed{\text{operaciones}}$$

  podríamos tener el *tipo de datos de los polinomios* al mismo nivel que los enteros o las cadenas de texto (como si C# los tuviese incorporados *de serie*)

  - e incluso usarlos para definir otros nuevos tipos de datos.
  - o extenderlos para obtener variantes del tipo.

- Otro programador podrá utilizar el tipo de los polinomios
  - sin necesidad de conocer los detalles de la representación e implementación, que serán partes privadas de ese tipo de datos (por ejemplo, no necesita conocer la ordenación de monomios *interna* que se maneja)
  - pero conociendo las operaciones públicas de los mismos (suma, evaluación, etc)
- Si necesitásemos una operación *especial* (atípica) sobre los polinomios, se podría definir otro tipo de datos que herede y extienda el tipo que ya tenemos sin cambiar este $\rightsquigarrow$ reutilización

Por otro lado:

- Hemos implementado la ordenación de monomios
- si necesitamos ordenar otro tipo de datos, reimplementamos la ordenación?
- $\rightsquigarrow$ podríamos tener una ordenación genérica de vectores y no reimplentar $\rightsquigarrow$ reutilización

(En realidad C# ya incorpora el método `Sort` para arrays de cualquier tipo, siempre que hay un orden definido para sus elementos)

Ideas previas de Programación Orientada a Objetos

- Los objetos NO añaden nueva funcionalidad a nuestros programas en sentido estricto: no añaden nuevas instrucciones ...
  - Ya conocemos (esencialmente) todas las instrucciones del lenguaje
  - Cualquier programa puede escribirse sin utilizar objetos
- ...pero ofrecen una solución al problema de diseño y organización de los programas.

Cuentas bancarias

Supongamos que queremos diseñar un programa para gestionar las cuentas de un banco.

Con lo que conocemos hasta ahora, podemos declarar:

```
namespace cuentasBancarias
{
  // estado de la cuenta
  enum EstadoCuenta {Activa, Cerrada, Bloqueada, EnAuditoria};

  // datos de la cuenta
  struct Cuenta {
    public int Numero; // identificador cuenta
    public int Saldo;
    public int LimiteDescubierto;
    public string Nombre;  // nombre cliente
    public string Direccion;  // direccion  cliente
    public EstadoCuenta Estado;  // estado de la cuenta
  }

  class MainClass
  {
     ...
```


Después, *dentro de la clase*, podemos definir las operaciones de manipulación de la cuenta:

```
class MainClass
{
  static Cuenta creaCuenta(int numero, int saldo, int limite,
                           string nombre, string direccion, EstadoCuenta estado){
    Cuenta c;
    c.Numero = numero;
    c.Saldo = saldo;
    c.LimiteDescubierto = limite;
    c.Nombre = nombre;
    c.Direccion = direccion;
    c.Estado = estado;
    return c;
  }

  static void ingresa(ref Cuenta c, int cantidad){
    c.Saldo += cantidad;
  }

  static void bloquea(ref Cuenta c){
    c.Estado = EstadoCuenta.Bloqueada;
  }

   ...
```

Los structs, además de *campos* para agrupar los datos pueden incluir

- métodos para manipular esos datos
- un método especial (*constructor*) para inicializar los campos

```
namespace cuentasBancarias
{
  // estado de la cuenta
  enum EstadoCuenta {Activa, Cerrada, Bloqueada, EnAuditoria};

  // datos de la cuenta
  struct Cuenta {
    public int Numero; // identificador cuenta
    public int Saldo;
    public int LimiteDescubierto;
    public string Nombre;  // nombre cliente
    public string Direccion;  // direccion  cliente
    public EstadoCuenta Estado;  // estado de la cuenta

    // constructor, mismo nombre que el struct
    public Cuenta(int numero, int saldo, int limite,
                  string nombre, string direccion, EstadoCuenta estado){
      Numero = numero;
      Saldo = saldo;
      LimiteDescubierto = limite;
      Nombre = nombre;
      Direccion = direccion;
      Estado = estado;
    }
     ...
```

Ahora los campos del struct se comportan como variables visibles dentro de los métodos.

Al declarar una cuenta, podríamos hacerlo del modo habitual

```
Cuenta c;
```

y después inicializar sus campos normalmente: podemos hacerlo así porque esos son campos "public" ⇝ accesibles desde fuera del propio struct.

Si los campos fuesen "private" no serían accesibles desde fuera del struct...

- pero se puede inicializar una cuenta con `new` ⇝ desencadena automáticamente una llamada al constructor:

```
Cuenta c=new Cuenta(1, 234, 30,"Luis","C/ Rioja, 3",EstadoCuenta.Activa);
```

Que se consigue con esto? la representación de los datos y el acceso a los mismos son privados (ocultos), pero podemos gestionar las cuentas a través de métodos que acceden a esos datos:

- el constructor, y los demás métodos, tienen acceso a los campos del `struct`

Público y privado

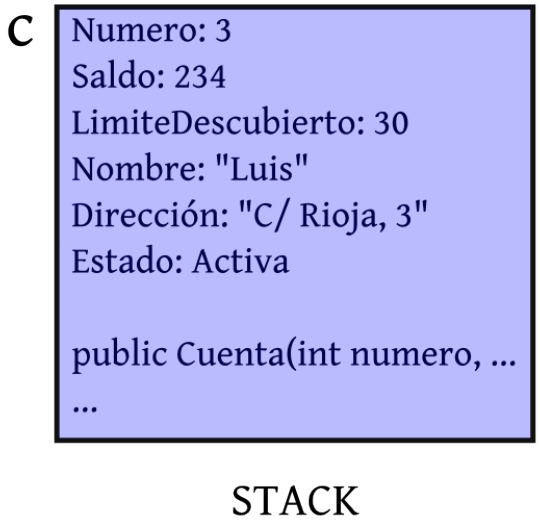
El ejemplo anterior proporciona una pista importante sobre la filosofía de la Programación Orientada a Objetos $\rightsquigarrow$ otra forma de abstracción:

- La representación de los datos y los datos propiamente dichos serán privados: inaccesibles desde fuera de la estructura
- Son accesibles *de modo indirecto* a través de métodos públicos que pueden ver y modificar esos datos.

La idea es una caja negra que contiene datos y métodos privados que los manipulan, y además un interfaz de métodos públicos para gestionar esos datos pero sin acceder directamente a ellos.

En memoria

La declaración y creación de una variable `c` de tipo `Cuenta` es similar a cualquier variable de tipo simple. Se almacena en memoria *estática* (STACK) y se representa con sus campos correspondientes.



Evolución clases: `struct` ⇝ `class`

```
using System;
namespace cuentasBancarias {
  class Cuenta {
    // estado de la cuenta
    private enum EstadoCuenta {Activa, Cerrada, Bloqueada, EnAuditoria};

    // "datos" miembro (atributos) de la clase accesibles a los métodos
    private int Numero; // identificador cuenta
    private int Saldo;
     ...

    public Cuenta(int numero, int saldo, int limite,    // constructor
                  string nombre, string direccion){
      Numero = numero;
      Saldo = saldo;
      LimiteDescubierto = limite;
      Nombre = nombre;
      Direccion = direccion;
      Estado = EstadoCuenta.Activa; // por defecto la dejamos activa
    }

    // los métodos pueden acceder a los "datos" de la clase,
    // no necesitan llevarlos como argumento
    public void ingresa(int cantidad){ Saldo += cantidad; }

    public int dameSaldo(){ return Saldo; }  // observadora

    // modificadora
    public void aumentaDescubierto(int incremento) { LimiteDescubierto += incremento; }

    public void bloquea(){ Estado = EstadoCuenta.Bloqueada; } // modificadora
  }
```


Ahora tenemos definida la clase `Cuenta` y podemos declarar e inicializar variables de ese tipo.

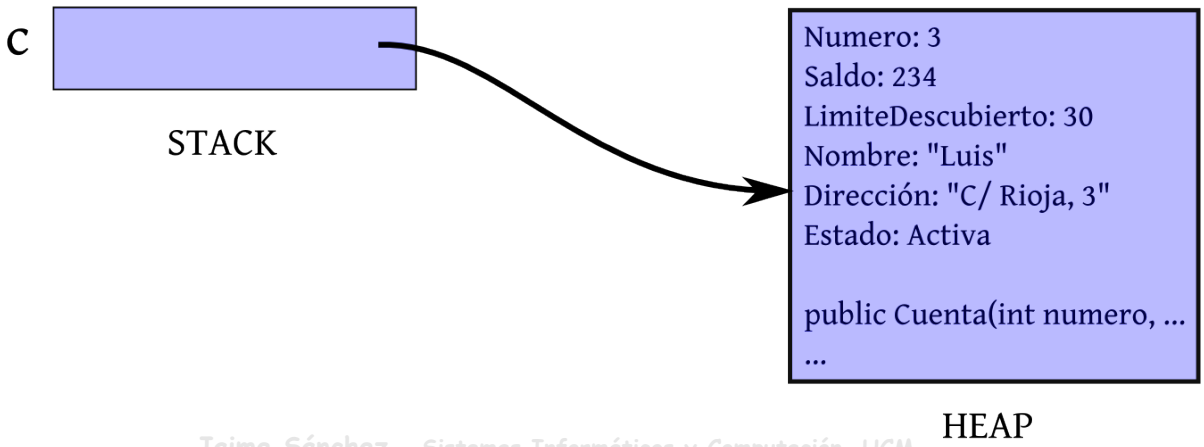
```
class MainClass
{
  public static void Main (string[] args)
  {
    // declaracion
    Cuenta c;
    //  inicializacion
    c = new Cuenta(123, 2000, 30, "Juan Pedro", "C/ Rioja, 22");

    c.ingresa(1000);
    c.bloquea();
     ...


    //c.Nombre = "Pepe";  error! no se permite acceso
```

Esto es todo?. . . Qué implicaciones tiene este cambio?

En memoria

Ahora también se puede declarar una variable `c` del tipo `Cuenta`. Pero al igual que con los arrays, hay que crear la estructura en memoria con `new`. Técnicamente:

- la variable `c` es una referencia (no es la cuenta en sí)
- a un objeto, una instancia de la clase `Cuenta`, que se crea con `new`
- el objeto se almacena en memoria dinámica (*HEAP*)



Referencias: por qué son tan importantes?

- Objetos y las referencias son dos conceptos inseparables en POO (al menos en lenguajes como C# o Java)

Trabajar con referencias tiene implicaciones importantes:

- Podemos tener múltiples referencias a la misma instancia
  - Distintos objetos pueden compartir memoria (datos) a través de las referencias
- Puede haber instancias sin ninguna referencia.

Múltiples referencias a la misma instancia

Supongamos que definida la clase `Cuenta` y el código siguiente:

```
namespace cuentasBancarias
{
  class Cuenta {
    ...
  }


  class MainClass
  {
    public static void Main (string[] args)
    {
      Cuenta c = new Cuenta(123, 2000, 30, "Juan Pedro", "C/ Rioja, 22");
      Console.WriteLine("Saldo c: " + c.dameSaldo());

      Cuenta c2 = c;
      c2.ingresa(1000);

      Console.WriteLine("Saldo c: " + c.dameSaldo());
    }
  }
}
```

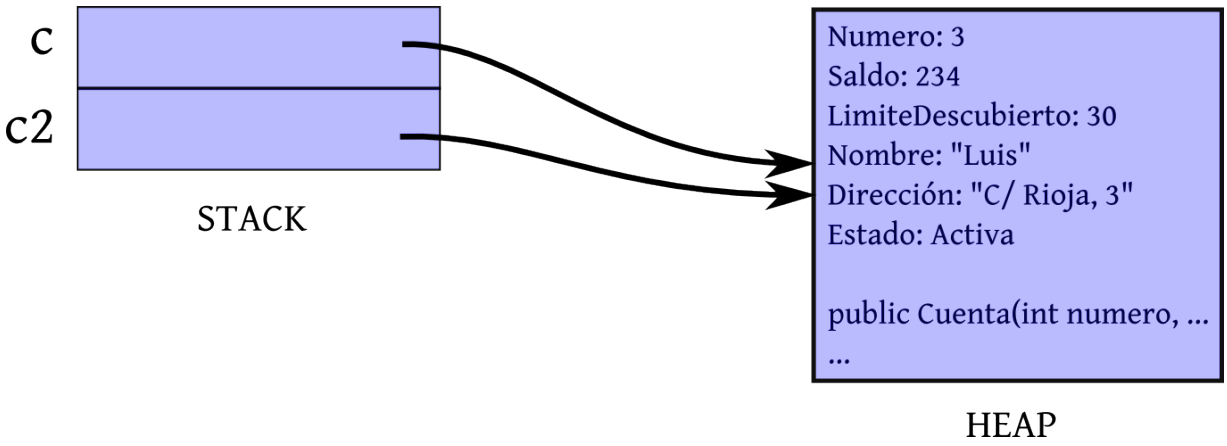
Qué muestra en pantalla?

```
Saldo c: 2000
Saldo c: 3000
```

Por qué?

En memoria

Tenemos dos referencias al mismo objeto! a

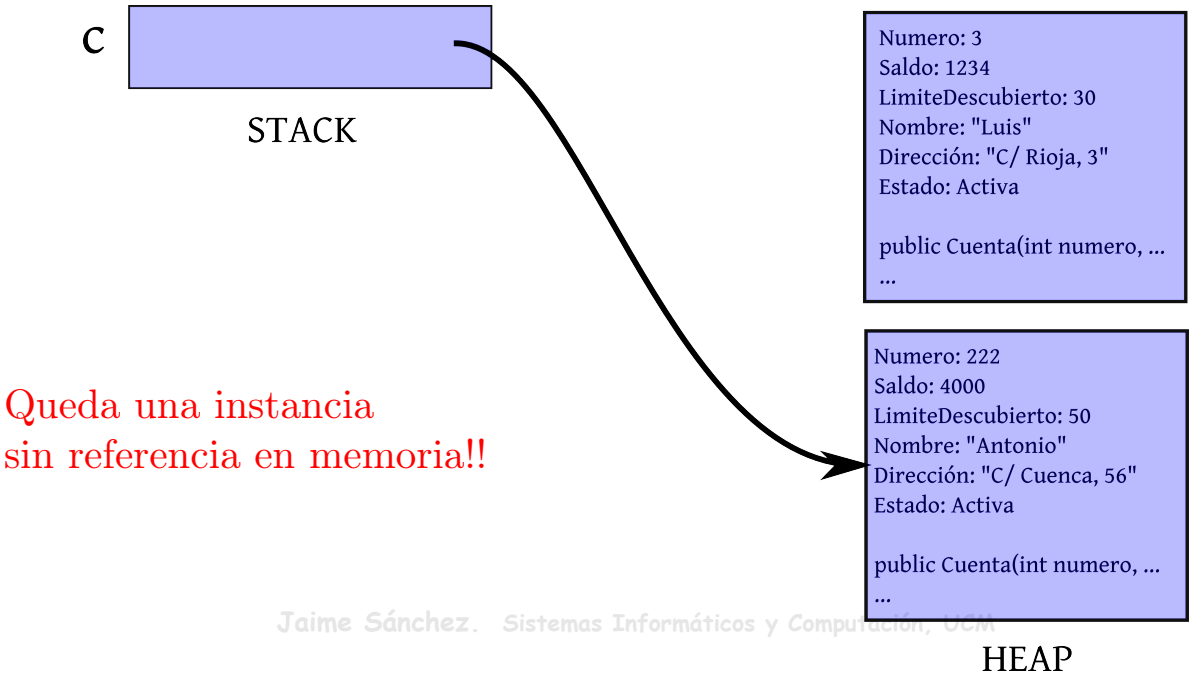


Los cambios sobre `c` se reflejan en `c2` (y viceversa).

Instancias sin ninguna referencia

Qué ocurre si hacemos lo siguiente:

```
class MainClass {
  public static void Main () {
    Cuenta c = new Cuenta(3, 234, 30, "Luis", "C/ Rioja, 3");
    c.ingresa (1000);
    c = new Cuenta(222, 4000, 50, "Antonio", "C/ Cuenca, 56");
```



Queda una instancia
sin referencia en memoria!!

Una forma más directa de conseguir el mismo efecto:

```
// bloque de código
{
    Cuenta c = new Cuenta(3, 234, 30, "Luis", "C/ Rioja, 3");
}
```

La variable `c` es local al bloque y desaparece al terminar este:

- la variable `c` desaparece (se elimina del *STACK*)
- pero la instancia creada en el *HEAP* no desaparece y además queda inaccesible en memoria: *es basura*

$\rightsquigarrow$ si vamos dejando basura podríamos llenar la memoria con basura

C# (y otros lenguajes como Java) han solucionado este problema de una forma simple y elegante:

- Incorporan un recolector de basura (garbage collector): proceso encargado de buscar instancias en el HEAP sin referenciar y eliminarlas para liberar la memoria correspondiente.
- Este proceso es transparente para nosotros: no tenemos que preocuparnos de ello.

Por qué trabajar con referencias?

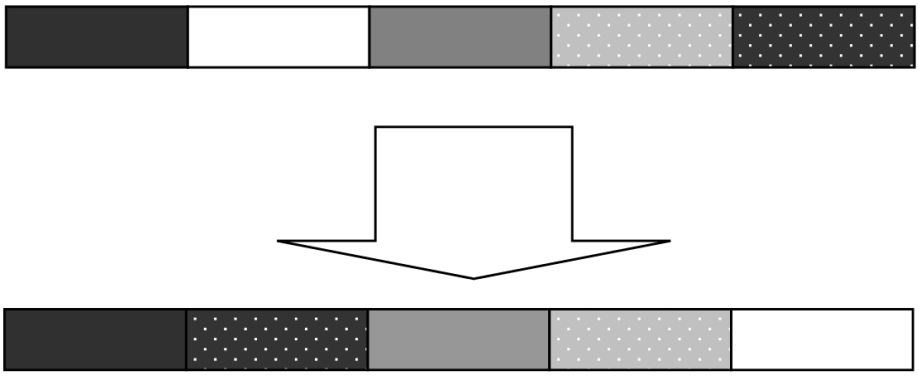
- Las referencias *complican* la vida al programador y al principio pueden crear cierto grado de confusión. . . por qué C# maneja referencias?

Imaginemos un banco:

- con miles (o millones) de cuentas de clientes
- cada cuenta con multitud de datos: datos personales del cliente, tarjetas asociadas, histórico de movimientos de la cuenta, . . .
- almacenamos las cuentas en un array

Y supongamos que queremos ordenar las cuentas de acuerdo a un criterio cualquiera (número de cuenta, saldo, nombre del cliente, . . .)

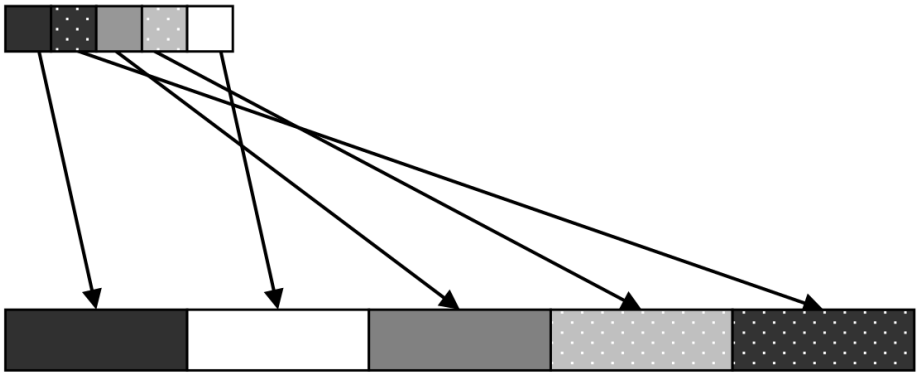
Tendríamos algo así (C# Programming Yellow Book, Rob Miles, "Bananas" Edition 7.0, 2015):



Para ordenar (con el algoritmo que utilicemos de ordenación) intercambiar o copiar una cuenta bancaria en una componente del array, supone copiar todos los datos de la cuenta (no es tan simple como intercambiar dos enteros) $\rightsquigarrow$ elevado coste computacional.

- Y si después nos piden la ordenación de acuerdo a otro criterio?
- Y si nos piden mantener simultáneamente las cuentas ordenadas con dos criterios?

Si utilizamos referencias podemos mantner un array de referencias ⇝ para ordenar lo que se intercambia son referencias y no los datos de las cuentas bancarias (C# Programming Yellow Book, Rob Miles, "Bananas" Edition 7.0, 2015):

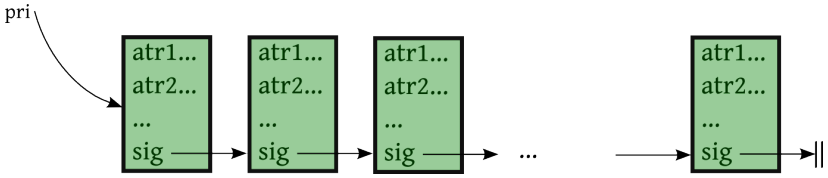


Podemos incluso mantener varios arrays de referencias, ordenados con distintos criterios de ordenación, i.e., mantener simultáneamente las cuentas ordenadas respecto a varios criterios de ordenación.

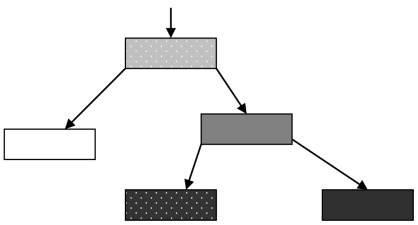
Referencias y estructuras de datos

Las referencias permiten diseñar estructuras de datos más sofisticadas:

- Listas enlazadas: los objetos incluyen un miembro *siguiente*, una referencia a al siguiente en la lista.



- Estructuras arbóreas:



- En general, estructuras tan complejas como se quiera...

Diseño con objetos

- Cambio de filosofía para el programador. Ahora:
  - no pensamos qué hacer con las cuentas de un banco
  - pedimos a las cuentas que hagan cosas para nosotros

  El concepto *cuenta* es más que un agregado de datos
- Determinar y controlar qué cosas puede y no puede hacer una cuenta (retirar efectivo por encima del límite permitido, etc). Especificar el comportamiento de la clase.
- *Proteger* los datos: hacerlos privados y proporcionar los métodos adecuados para manipularlos.

Si hacemos:

```
class Cuenta {
   ...
  private int Saldo;

}

class MainClass  {
  public static void Main () {
    Cuenta c = new Cuenta(123, 2000, 30, "Juan Pedro", "C/ Rioja, 22");
    c.Saldo = 0;
```

Obtenemos un error en compilación:

```
Error CS0122:  'cuentasBancarias.Cuenta.Saldo' is
inaccessible due to its protection level (CS0122)
```

La modificación de miembros privados de una clase se hará a través de métodos públicos de la clase $\rightsquigarrow$ protocolo de comunicación con la clase (accesos controlados, seguridad).

En general:

- Los datos miembro del objeto se declararán privados
- Los métodos de la clase serán públicos
  - salvo aquellos que se implementen como auxiliares para los anteriores

. . . hay más posibles declaraciones para los miembros de una clase

Datos miembro `static`

Dato adicional en las cuentas bancarias: comisión de mantenimiento (cargo anual en concepto de mantenimiento de la cuenta).

```
class Cuenta {
  ...
  private int comisionMantenimiento;
  ...
}
```

Esta comisión es la misma para todas las cuentas . . . si cambia en un momento dado, habría que cambiarla en todas las cuentas (en todos los objetos de la clase `Cuenta`)

- Este dato de manera natural pertenece a la clase y no a cada instancia de ella $\rightsquigarrow$ debería ser miembro de la clase y no de cada instancia (objeto).

Esto se consigue con la declaración static

```
private static int comisionMantenimiento;
```

Ahora se podría implementar un método para modificar este dato y tenemos el efecto buscado: la comisión cambia para todas las cuentas.

Métodos `static`

Nueva restricción: hay que ser mayor de edad e ingresar al menos 1000 euros para abrir una cuenta.

Situación paradójica: antes de abrir una cuenta (crear una instancia) hay que hacer esta comprobación . . . que se hace con un método de esa instancia!! $\rightsquigarrow$ la *operación* (método) es propia de la clase y no de una instancia concreta.

```
public static bool cuentaPermitida(int edad, int cantidad){
  return ((edad>=18) && (cantidad>=1000));
}
```

Ahora, desde fuera de la clase se puede hacer:

```
if (Cuenta.cuentaPermitida(19,2000)) Console.WriteLine("Cuenta permitida");
else Console.WriteLine("Error: cuenta no permitida");
```

Sin llegar a crear la cuenta!!

Observación: el método `Main` siempre es estático. . . por qué? : no tiene sentido tener varios métodos `Main` (instanciando la clase que lo contiene) y además se necesita llamar a `Main` al principio, antes de tener ninguna instancia de clase.

Mejorando el diseño (métodos para modificar `edadMinima` y `minimoApertura`):

```
class Cuenta {
  ...
  // inicialización "por defecto"
  private int edadMinima = 18;
  private int minimoApertura = 1000;

  public static bool cuentaPermitida(int edad, int cantidad){
    return ((edad>=edadMinima) && (cantidad>=minimoApertura));
  }

  // modificacion
  public static void setEdadMinima(int edad) { edadMinima = edad;}
  public static void setMinimoApertura(int min) { minimoApertura = min; }
  ...
```

Por qué está mal? Hay una incoherencia en las declaraciones

```
Error CS0120: An object reference is required to access
non-static member 'cuentasBancarias.Cuenta.edadMinima'
```

Un método estático no puede modificar un dato no estático (que está en cada instancia de la clase). Debería ser:

```
  private static int edadMinima = 18;
  private static int minimoApertura = 1000;
```

(Métodos) constructores revisitados

Hemos visto un método especial: el constructor de la clase.

```
Cuenta c; // declaración
c = new Cuenta(1, 234, 30,"Luis","C/ Rioja, 3"); // creación (inicialización)
```

`new` solicita hueco en el *heap* para un objeto `Cuenta` e invoca automáticamente al constructor de la clase.

- Cada clase tiene que tener un método constructor.
- El nombre es siempre el mismo que el de la clase.
- Se invoca siempre al crear un objeto (o instancia) de esa clase.
- Este método es público y no devuelve nada.
- El constructor debe inicializar los miembros del objeto creado.

Pero C# no exige que implementemos un método constructor!!
$\rightsquigarrow$ si no lo implementamos C# automáticamente lo hace por nosotros, generando un método vacío (sin código):

```
public Cuenta(){}
```


Lo más habitual es que el programador implemente el constructor tal como hemos hecho antes:

```
public Cuenta(int numero,int saldo,int limite,string nombre,string direccion){
  Numero = numero;
  Saldo = saldo;
  LimiteDescubierto = limite;
  Nombre = nombre;
  Direccion = direccion;
  Estado = EstadoCuenta.Activa; // por defecto la dejamos activa
}
```

De este modo, al crear una cuenta tenemos que proporcionar obligatoriamente el número, el saldo, etc. Ahora no se puede invocar a este constructor sin argumentos $\rightsquigarrow$ control al crear cuentas.

Hay mucho más que aprender sobre clases y POO en general: interfaces, herencia (simple o múltiple), sobrecarga de métodos, polimorfismo, componentes, ...
Pero no lo abarcamos en este curso.