**COMPLEJIDAD DEL SOFTWARE**

**Tipos de software:**

* Software no complejo.
* Software complejo.

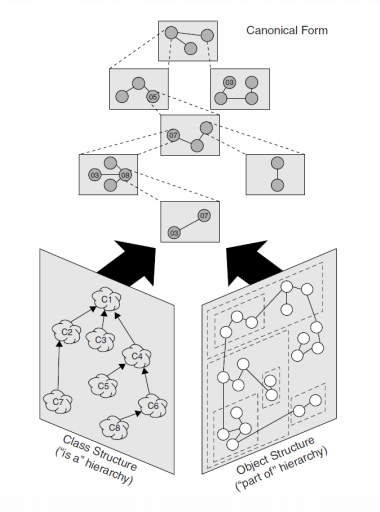
¿**Por qué el software es complejo?**

* **Complejidad del dominio del problema**
  + Muchos requerimientos.
  + Communication gap entre programadores y usuarios.
  + Los requerimientos cambian durante el proceso.
* **Dificultad para gestionar el proceso de desarrollo**
  + El software crece hasta un punto donde se necesita un equipo de desarrollo.
  + El equipo demanda trabajo extra (comunicación, integridad de diseño).
* **Flexibilidad alcanzable gracias al software**
* **Problemas de caracterizar el comportamiento de sistemas discretos**
  + Pequeñas variaciones en la entrada pueden producir grandes variaciones en las salidas.
  + Necesidad de testing.

**Atributos de sistemas complejos**

* **Separación de preocupaciones**
  + El software puede ser modularizado, aunque los módulos tengan dependencias entre ellos.
  + Los vínculos internos de un módulo son más fuertes que los externos módulos.
  + Permite el estudio de los módulos por separado.
* **Patrones comunes**
  + Permite reutilizar pequeños componentes que forman partes de diversos sistemas.
* **Formas intermedias estables**
  + Los sistemas tienen a evolucionar.
  + En la mayoría de los casos, los sistemas complejos son evoluciones de sistemas más simples que funcionaban.
  + Es muy difícil producir un sistema complejo desde cero.

**Forma canónica de un sistema complejo**

* **Jerarquía “parte de” ⬄ estructura de objetos**
  + Representa los objetos que interactúan en el sistema.
* **Jerarquía “es un” ⬄ estructura de clases**
  + Captura propiedades comunes de los objetos.
* En ambas jerarquías, los niveles muestran el nivel de primitivas del sistema.
* El conjunto de ambas jerarquías se denomina “arquitectura”.

**Limitaciones de las personas para lidiar con la complejidad**

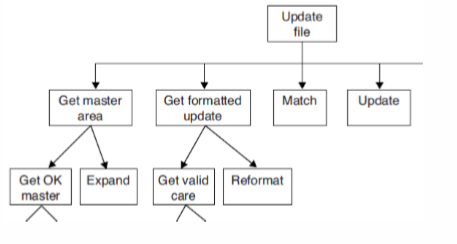
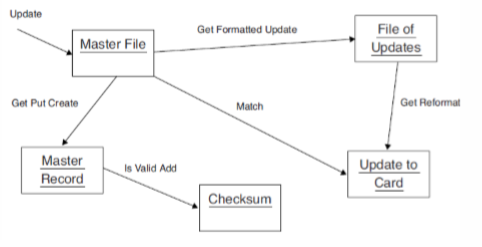
Si sabemos cómo lidiar con sistemas complejos, ¿Por qué seguimos teniendo problemas en desarrollarlos?

Los sistemas aumentan en complejidad pero las personas solo pueden lidiar con 5 +/- 2 cosas a la vez.

**La pregunta a responder es: ¿Cómo podemos resolver este dilema?**

* Descomposición.
* Abstracción.
* Jerarquías.

**Descomposición (divide et impera)**

* Descomponer un sistema en subsistemas hasta que estos sean lo suficientemente pequeños como para entenderlos.
* **Descomposición algorítmica**
  + Cada módulo consiste en un paso de un proceso que lo contiene.
* **Descomposición orientada a objetos**
  + Descomposición en objetos que se derivan directamente del vocabulario del problema. Se ve el mundo como un conjunto de objetos autónomas que colaboran entre sí para lograr un objetivo.

**Programación orientada a objetos (POO)**

La programación orientada a objetos es un método de implementación en la cual los programas se organizan como colecciones colaborativas de objetos, cada uno de los cuales representan una instancia de alguna clase, donde todas las clases son miembros de una jerarquía de clases unidas por relaciones de herencia.

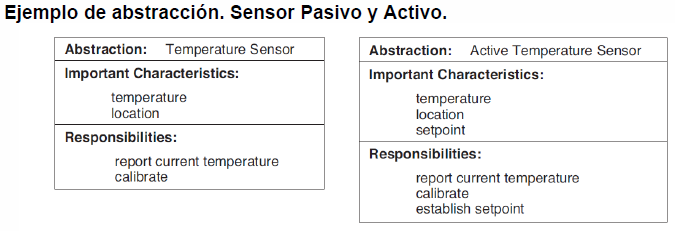
* Usan objetos como bloque básico de construcción.
* Cada objeto pertenece a una clase.
* Las clases se relacionan por herencias.

**Elementos del modelo de objetos**

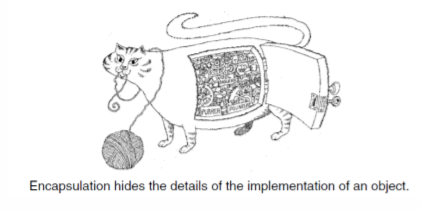
* Elementos principales (necesarios):
  + Abstracción
  + Encapsulamiento
  + Modularidad
  + Jerarquías
* Elementos secundarios (útiles pero no esenciales)
  + Typing
  + Concurrency
  + Persistence

**Abstracción**

* Una abstracción representa las características esenciales de un objeto que permiten distinguirlo de todos los otros tipos de objetos. Este conjunto de características proporcionan fronteras conceptuales claramente definidas, relativas a la perspectiva del observador.
* Una abstracción se centra en la vista externa de un objeto, por lo tanto permite separar el comportamiento esencial de un objeto de su implementación.
* Nos vamos a centrar en la abstracción de entidades porque se deduce directamente del dominio del problema.



**Encapsulamiento**

* El encapsulamiento se refiere al ocultamiento de toda información que no expresa las características esenciales de una abstracción.
* Abstracción y encapsulamiento serian conceptos complementarios
  + Abstracción se focaliza en el comportamiento observable de un objeto.
  + Encapsulamiento se focaliza en ocultar la implementación o los mecanismos que materializan esos comportamientos.

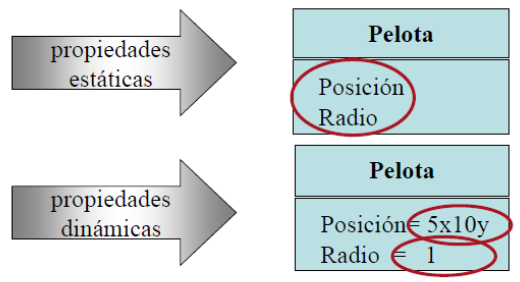
**Modularidad**

* Modularizar consiste en dividir un programa en unidades que pueden ser compiladas independientemente, módulos, pero que mantienen conexiones entre sí.
* Las conexiones entre módulos son las suposiciones que cada módulo hace sobre los restantes.

**Jerarquías**

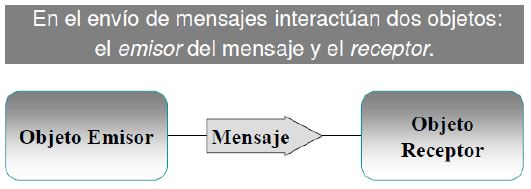
* Nos permiten pensar en varias abstracciones como si fueran una sola.
* Las jerarquías nos permiten ordenar las abstracciones.

**¿Qué es (y no es) un objeto?**

* **Definición de objeto**
  + Es una entidad que posee estado, comportamiento e identidad. El estado y comportamiento de objetos similares están definidos en sus clases. El término “instancia” y “objeto” tiene el mismo significado.
* Un objeto representa un individuo, un ítem identificable, una unidad o una entidad, real o abstracta, con un rol bien definido en el dominio del problema.
* Ejemplos de objetos: Una factura, un proceso, una figura, etc.
* Hay que tener cuidado de no identificar como objetos cosas que son propiedades de los objetos, por ejemplo: Colores, sentimientos, emociones.
* **Estado**
  + El estado de un objeto abarca todas las propiedades (usualmente estáticas) del objeto más los valores actuales (usualmente dinámicos) de cada una de esas propiedades.
* **Comportamiento**
  + El comportamiento es como un objeto actua y reacciona, en termino de su estado y el paso de mensajes.
  + El estado de un objeto representa la acumulacion de resultados del comportamiento.
  + Las opèraciones se denominan metodos:
    - Modificadores: Alteran el estado del objeto.
    - Selectores: Acceden al estado de un objeto pero no lo alteran.
    - Iteradores: Permiten acceder a todas las partes de un objeto en un orden determinado.
    - Constructores: Crea o inicializa el objeto.
    - Destructores: Libera el estado de un objeto y lo destruye.
* **Protocolo**
  + El protocolo de un objeto define el flujo permitido de mensajes que un objeto puede recibir.
* **Identidad**
  + Es la propiedad que distingue un objeto del resto de los objetos.

**¿Cómo interactúan los objetos entre sí?**

* Los objetos contribuyen al comportamiento de un sistema colaborando unos con otros.
* Todas las operaciones en la POO resultan de enviar mensajes entre objetos.



* Un mensaje es una solicitud para que un objeto ejecute una acción. Este objeto responde ejecutando métodos, es decir, usando sus propios procedimientos. El paso de mensajes es sincrónico.
* El objeto emisor debe conoicer la identidad del objeto receptor. La puede obtenber de alguna de las siguientes maneras:
  + Estado objeto emisor.
  + Argumento del metodo desde donde se emite el mensaje.
  + Retorno del envio de un mensaje previo.
* **El envio de mensajes soporta un importante principio: La abstraccion**
  + El estado interno de un objeto solo puede modificarse mediante el envio de algun mensaje valido.
  + Esto implica que puede cambiarse la implementacion de un objeto sin que varie el compotamiento del mismo, ni el programa que lo contiene.

**La naturaleza de las clases**

* Una clase representa solo la abstracción de un objeto, o sea, las características comunes a todos los objetos de esa clase.
* Una clase es un conjunto de objetos que comparten la estructura, comportamiento y semántica.

**Interfaz e implementación**

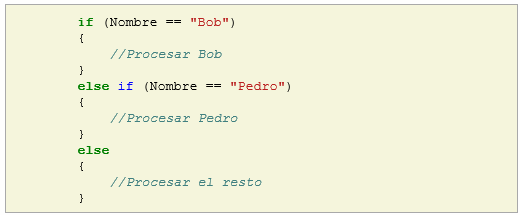
* La programación es una cuestión de contratos.
* Una clase sirve de vínculo entre una abstracción y todos sus clientes.
* Una clase se puede dividir en la vista externa e interna.
  + La vista externa es la interfaz, una declaración de todas las operaciones y propiedades disponibles en una clase.
  + La vista interna es la implementación, contiene todos los detalles de implementación para llevar a cabo las operaciones definidas en la interfaz.

**Introducción a C# y .Net**

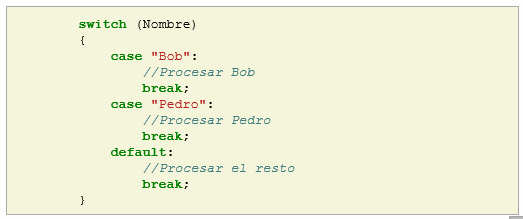
* **Tipos de datos**
  + **Por valor**: contienen directamente los datos. Se almacenan en el stack. Ej: System.Int32, System.Boolean.
  + **Por referencia**: contienen punteros a los datos almacenados en el heap. Ej: System.String, clases, colecciones.
* **Variables**
  + En C# se definen de la siguiente manera:



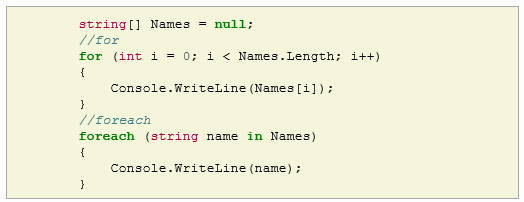
* **Instrucciones condicionales**
  + **if**



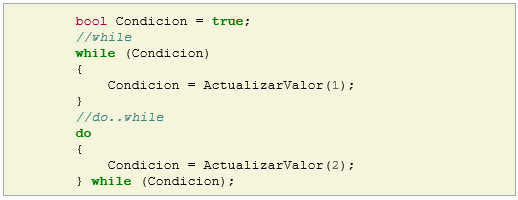
* + **switch**

****

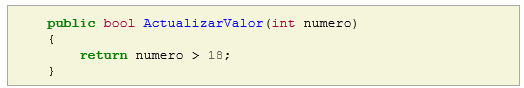
* **Bucles**
  + **For**



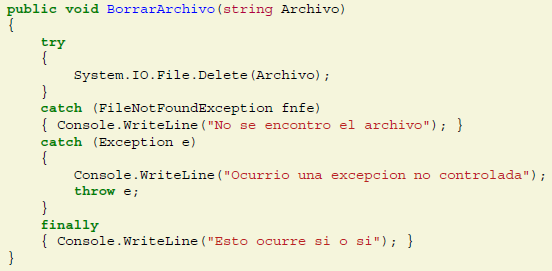
* + **while**



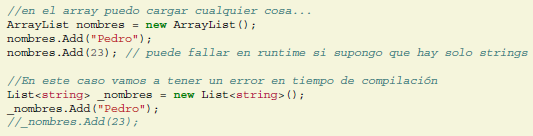
* **Funciones**



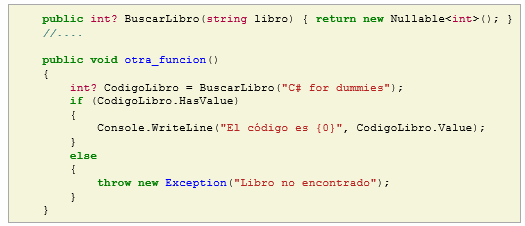
* **Manejo de excepciones**



* **Tipos genericos**



* + En el namespace **System.Collections.Generic** tengo disponibles:
    - Lista
    - Diccionario
    - Stack, Queue…
* **Nullables**
  + Permiten representar la ausencia de valor en tipos por valor.

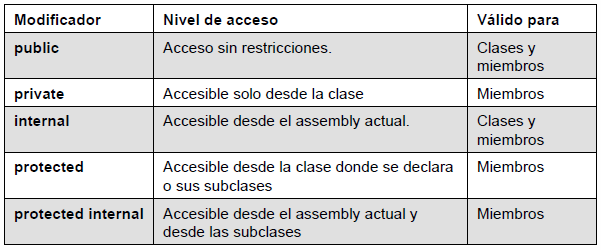


**Clases en C#**

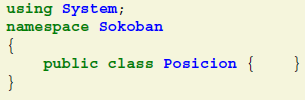
Algunos conceptos

* **Campos**: Representan datos en un objeto.
* **Propiedades**: Proveen acceso a elementos de un objeto.
* **Métodos**: Definen las acciones que un objeto puede hacer.
* **Eventos**: acciones que se pueden responder o manejar en el código.
* **Overloading** (Sobrecarga): Capacidad de tener varios métodos con el mismo nombre.

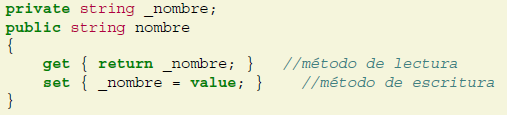
**Modificadores de acceso:**



**Definición de una clase:**

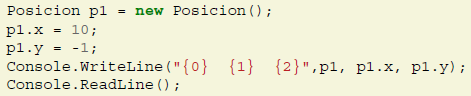


**Agregado de campos y propiedades:**

* Los campos son variables a nivel de instancia.
* Las propiedades nos permiten encapsular el estado. Pueden ser:
* **Solo lectura:** proveemos solo el método **get**
* **Solo escritura:** proveemos solo el método **set**
* **Lectura y escritura:** proveemos ambos métodos

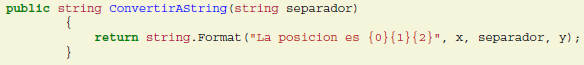
**Creación de instancias**

Para crear una instancia definimos una variable de ese tipo y utilizamos la palabra **new.**



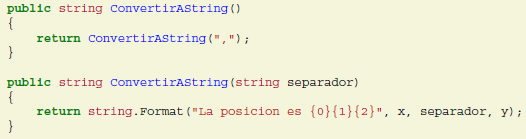
**Agregando métodos**

Los métodos se declaran dentro de la clase de la siguiente manera:



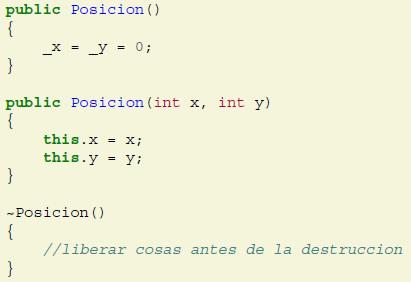
**Sobrecarga (overloading)**

* Capacidad para tener dos o más métodos con el mismo nombre.
* Los métodos se tienen que diferenciar por su firma (cantidad y tipo de parámetros).



**Agregando constructores y destructores**

* **Un constructor es un método que se ejecuta ni bien termina la creación de un objeto. Sirve para inicializar la instancia en cuestión.**
* Tienen el mismo nombre que la clase en la que están definidos.
* No deben especificar un valor de retorno.
* Se pueden definir varios constructores sobrecargados. El constructor que no acepta parámetros es el constructor por default.
* **Un destructor es un método que se ejecuta cuando el objeto es recolectado por el garbage collector.**
* Tienen el mismo nombre que la clase en la que están definidos, precedido por un **~.**
* No debe especificar un valor de retorno ni tomar parámetros.



**Agregando miembros de clase**

Todos los miembros mencionados hasta el momento son de instancia. Estos pertenecen a una instancia y deben ser accedidos por medio de ella.

Una clase puede contener miembros de clase, los cuales son accedidos sin la necesidad de crear una instancia.

Para definir los miembros de clase hacemos uso de la palabra clave static.

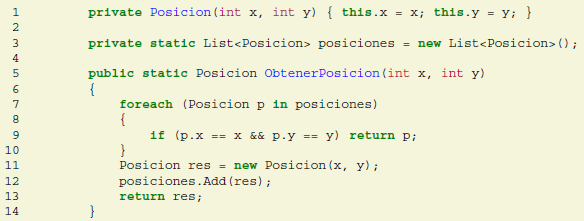
* Ejemplo: ¿Qué pasa si tenemos 2 instancias de posición con las mismas coordenadas y las comparamos?



¿Cómo podriamos remediar esto?

* Usando una variante del patrón *singleton*
* Cambiar la forma de comparar las instancias.

El patrón singleton asegura que para una clase haya solo una instancia. En este caso vamos a aplicar el patrón con la salvedad que hay varias instancias dependiendo del estado interno del objeto.



1. Hacemos inaccesible el constructor desde fuera de la instancia

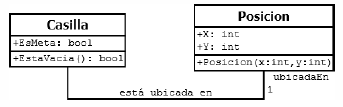
3. Agregamos un campo de clase para almacenar las instancias que se van creando

5. Agregamos un método para obtener una instancia, el cual revisa si la instancia buscada ya fue creada para no volverla a crear.

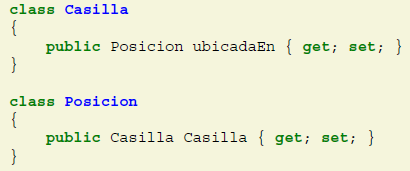


**Relaciones entre clases**

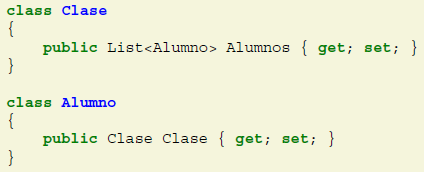
* Las relaciones más comunes entre clases son asociación, agregación, composición y herencia:
* **Asociación**
  + Es la relación más general, la que menos semántica lleva asociada.
  + Denota la dependencia entre dos clases.
  + La representación es UML es una línea.



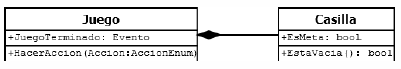
* + Cuando programamos dos clases relacionadas mediante asociación, deberíamos colocar un campo en cada clase con el endpoint de la relación para tener una referencia a la otra clase. Hay que tener en cuenta la cardinalidad para saber si solo necesitamos una variable o una lista.

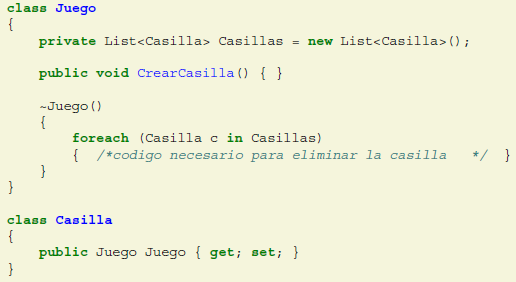


* **Agregación**
  + Provee la semantida de Todo/Parte de.
  + No se realiza ninguna vinculacion acerca del ciclo de vida de las clases vinculadas.
  + La cardinalidad de la relacion generalmente es de 1 a muchos.



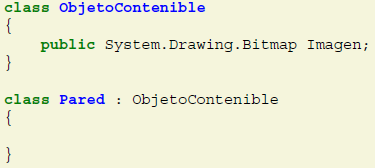
* **Composición**
  + Es una variación de la agregación.
  + Expresa una relación de dependencia y asocia los ciclos de vida de las pares con el del todo.
  + El todo es responsable del ciclo de vida de las partes, lo cual quiere decir, administrar la creación y destrucción.





* **Herencia**
  + Permite expresar relaciones de generalización/especialización.
  + Una clase B que hereda de una clase A tiene todas las caracteristicas de A mas las caracteristicas propias.
  + La herencia puede ser:
    - Simple: Toda clase hereda de una sola clase
    - Múltiple: una clase puede tener mas de una clase padre

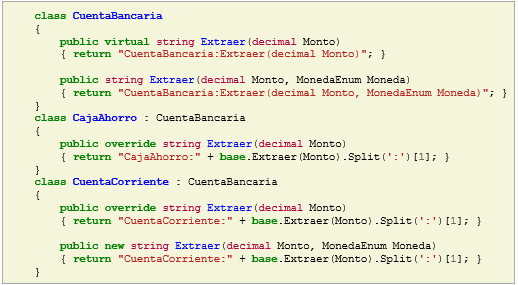




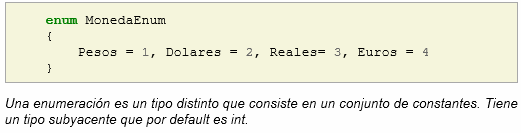
**Overriding – Hiding**

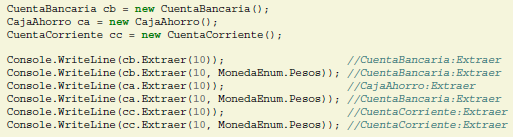
* Es una técnica que permite redefinir alguna propiedad o implementación de método en subclases
* Para hacer uso de la misma, en la clase base hay que definir el miembro como sobreescribible
* En la subclase se puede:
  + Derivar los miembros de la clase base. Esto ocurre por defecto.
  + Sobreescribir estos miembros.
  + Ocultar los miembros.

**Override vs Hide**

* Sobreescribir miembros
  + En la clase base los miembros deben tener el modificador virtual.
  + En la clase derivada el miembro es marcado con la palabra override.
  + El miembro está sujeto a las reglas del polimorfismo.
* Ocultar miembros
  + No hace falta que los miembros en la clase base tengan el modificador virtual.
  + En la clase derivada el miembro es marcado con la palabra new.
  + El miembro no está sujeto a las reglas del polimorfismo.

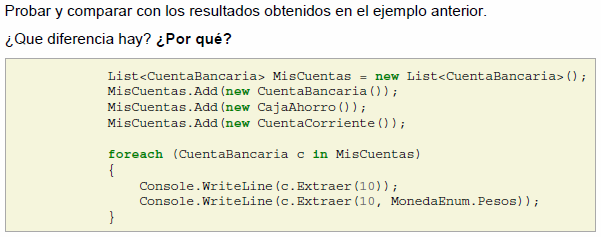
**Enumeraciones**

****



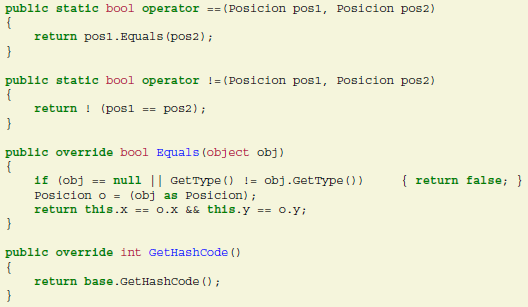
**Polimorfismo**

* Permite que una aplicación cree una instancia de una clase derivada y la asigne a una variable de la clase base.
* Si la aplicación invoca a un método sobreescrito a través de esta variable, el polimorfismo asegura que se ejecute la versión correcta del miembro invocado.
* Beneficios:
  + Consistencia y simplicidad: la aplicación puede lidiar con instancias de distintas subclases de una manera consistente.
  + Extensibilidad y resiliencia: si se agregan nuevas subclases el trabajo necesario para mantener el código es mínimo.



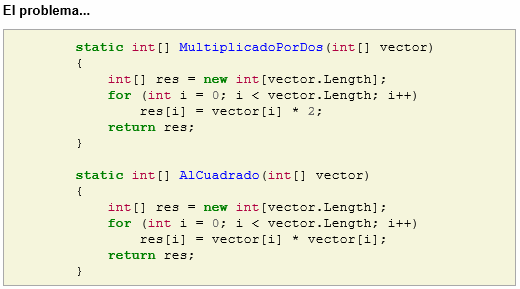
**Sobrecarga de operadores**

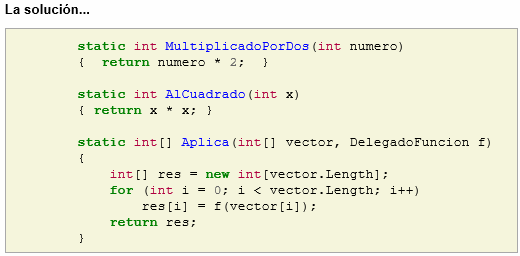
* Permite definir el comportamiento de los operadores del lenguaje cuando interactúan con nuestras clases.



**Delegados**

* Son objetos que encapsulan una referencia a un método de una clase u objeto.
* Para su uso hay que:
  1. El primer paso para usar un delegado es definir un tipo.
  2. Luego se pueden crear instancias del delegado y relacionarlas a métodos específicos.
  3. Por último, se puede invocar al método relacionado por medio del delegado.





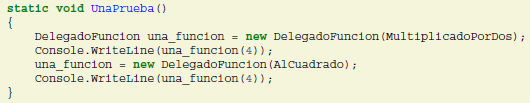
**Definición de un tipo de delegado**

* Se utiliza la palabra clave delegate.
* Hay que especificar la firma y el tipo de retorno del delegado.



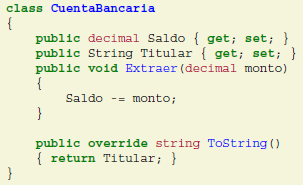
**Creación y uso de un delegado**

* Se declaran variables o parámetros como un tipo de dato más.
* Se los invoca como si fueran una función.

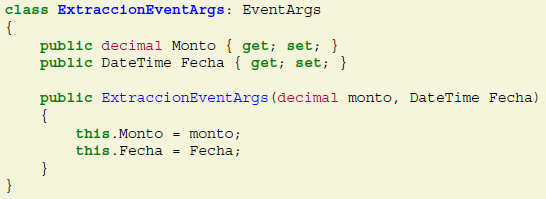


**Eventos**

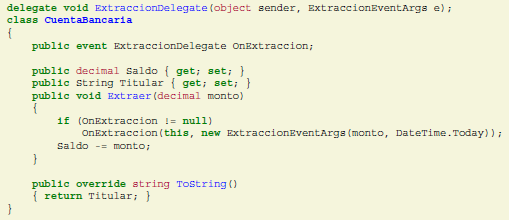
* Las clases y objetos pueden disparar eventos para avisar que algo importante ha ocurrido.
* El evento es el mecanismo por el cual un objeto le indica a otro que invoque determinado método si ocurre determinado suceso.
* Para indicar que método debe invocar el emisor del evento, se necesita definir un delegado.
* La manera estándar de pasar datos a un evento en .Net es definir una clase derivada de EventArgs.
* **Ejemplo**: Definir un evento para que una cuenta bancaria avise cuando se produce una extracción. Nos interesa saber la fecha y monto de la misma.



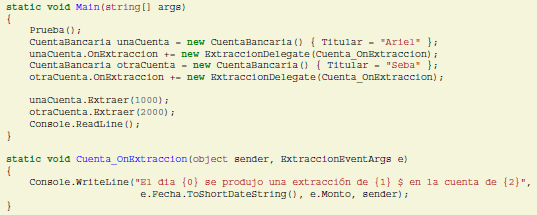
**Definiendo un derivado de EventArgs**



**Definición del delegado, el evento y disparo del mismo**

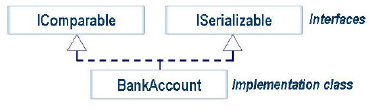


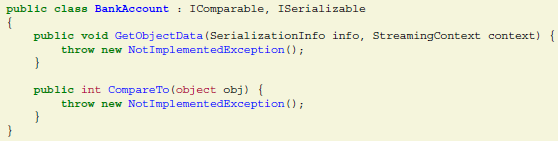
**Manejo del evento por otra clase**



**Interfaces**

* La interfaz de una clase define el comportamiento de dicha clase.
* Una interfaz define un conjunto de propiedades y métodos a ser implementados por otras clases.
* A diferencia de las clases, las interfaces no proporcionan implementación.
* Una interfaz representa un contrato.
* Las interfaces son implementadas por clases y se definen en entidades separadas.

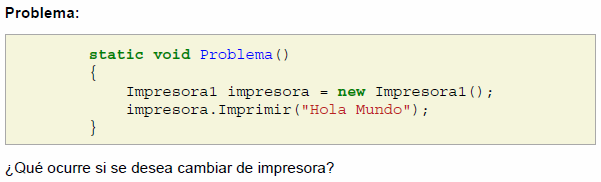


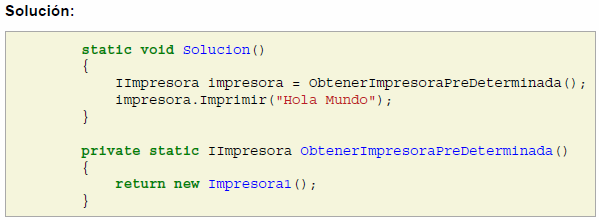


**Beneficios:**

* Desacoplamiento.
* Gestión de dependencias.
* Herencia múltiple de interfaces.

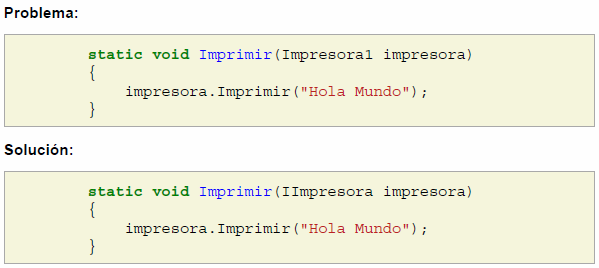
**Desacoplamiento**

* Las interfaces permiten desacoplar la aplicación de las implementaciones concretas de las clases.
* En la aplicación podemos referirnos a un tipo de interfaz en lugar de una clase exacta y de esta forma trabajar con cualquier clase que implemente la interfaz.



**Gestión de dependencias**

* Las interfaces permiten reducir las dependencias de la aplicación.
* Las interfaces son generalmente más estables que las clases que los implementan.



**Herencia múltiple de interfaces**

* Una clase puede heredar de varias interfaces.
* La clase debe implementar todos los miembros definidos en cada una de las interfaces que hereda.

