Lenguaje C# (Prog\_Lab2)

1. Conceptos Basicos

1.1. Tipos de Datos

Reference Types: Estos tipos son representados por una referencia a la ubicación en memoria del valor actual del objeto, similar a un puntero en C. Si asignan un Reference Type a una variable y luego la pasan a una función, cualquier cambio en el objeto se verá reflejado. NO se genera una copia.

Value Types: Estos tipos son representados por sus valores. Si asignan un Value Type a una variable se copia el valor.

Variables Escalares: Las variables escalares son constantes o variables que contienen un dato atómico y unidimensional.

Variables No Escalares: Las variables no escalares son array (vector), lista y objeto, que pueden tener almacenado en su estructura más de un valor.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Categoría** | **Clase** | **Descripción** | **C# Alias** | **Rango de valores** |
| **Enteros** | **Byte** | **Un entero sin signo (8-bit)** | **byte** | **0 a 255** |
|  | **SByte** | **Un entero con signo (8-bit)** | **sbyte** | **-128 a 127** |
|  | **Int16** | **Un entero con signo (16-bit)** | **short** | **-32768-32767/0 -65535** |
|  | **Int32** | **Un entero con signo (32-bit)** | **int** | **-2147483648- 2147483647/ 0-4294967295** |
|  | **Int64** | **Un entero con signo (64-bit)** | **long** |  |
| **Punto Flotante** | **Single** | **Un número de punto flotante de simple precisión (32-bit)** | **float** | **F** |
|  | **Double** | **Un número de punto flotante de doble precisión (64-bit)** | **double** | **D** |
|  | **Decimal** | **Un número decimal de 96-bit (con .)** | **decimal** | **M** |
| **Lógicos** | **Boolean** | **Un valor booleano (true o false)** | **bool** |  |
| **Otros** | **Char** | **Un caracter Unicode (16-bit)** | **char** |  |
|  | **Object** | **La raíz de la jerarquía de objetos** | **object** |  |
|  | **String** | **Cadena de caracteres unicode inmutable y de tamaño fijo** | **string** |  |

Valores Predeterminados

• Enteros, Punto flotante : 0 (cero)

• Lógicos: False

• Referencias: Null

Método: default(tipo)

Conversiones Básicas

• Implícitas: no interviene el programador. De un tipo menor a uno mayor.

• Explícitas: interviene el programador, ya que puede haber perdida de datos. Casteo

1.2. Formato de salida de Texto

{ N [, M ][: Formato ] }

• N: número del parámetro, empezando por cero. No es necesario que estén en orden.

• M: ancho usado para mostrar el parámetro, el cual se rellenará con espacios. Si M es negativo> justificado a la izquierda, positivo> justificado a la derecha. Tabulado.

• Formato: cadena que indicará un formato extra a usar con ese parámetro.

C (currency), D (decimal), E (científico), Fn (float+decimales), G (general), Nn(numero+decimales), X (hexa), P (%)

0 Se reemplazan los 0 por los dígitos indicado, si no hay se imprime 0

# Se reemplazan los # por los dígitos indicado, si no se imprime nada

. Separador decimal

, Separador miles

% multiplica por 100 + %

%. Multiplica por mil con separador de miles

/ el próx carácter es un literal

‘string’ el string se imprime sin modificación

; Separa secciones con distinto formato

dd/MM/yyyy fecha

Secuencias de escape:

@”” > todo el “” se interpreta como string

“” “” > p/imprimir comillas se doblan

1.3. Operadores aritméticos, lógicos + Sentencias condicionales e iterativas

Igual que en C salvo por foreach.

La sentencia foreach permite recorrer arreglos y colecciones

clase[] objeto = new clase[N];

foreach (clase elemento in objeto)

{ //auxNombre es un elemento de nombres. }

Switch:

case '1': goto case '0';

en C se puede dejar que un caso sea manejado por el siguiente, lo que se consigue si no se usa "break", mientras que C# siempre obliga a usar "break" o "goto" al final de cada cada caso, con la única excepción de que un caso no haga absolutamente nada que no sea dejar pasar el control al siguiente caso, y en ese caso se puede dejar totalmente vacío.

En el lenguaje C, que es más antiguo, sólo se podía usar "switch" para comprobar valores de variables "simples" (numéricas y caracteres); en C#, que es un lenguaje más evolucionado, se puede usar también para comprobar valores de cadenas de texto ("strings").

No necesito poner las firmas de las funciones antes de usarlas

1.4. Tipo por referencia

Convierten al parámetro formal en un alias para el argumento, que tiene que ser una variable. La palabra reservada se indica tanto en el argumento como en el param.

**in**: no permite que el método llamado modifique el valor del argumento. Tiene que estar inicializado. No es necesario en el llamado, solo en la declaración del método.

**ref**: la variable debe ser inicializada antes de pasarla. No es obligatorio modificar el valor en el método. Parámetro por referencia.

**out**: la variable no necesita estar inicializada antes de pasarla como argumento. El método llamado debe asignarle un valor antes del return. Parámetro de salida.

2. POO

2.1. Solución > Proyectos > NameSpaces > Clases > atributos y métodos > objetos

2.1.1. **Clase**:

[*modificador*] class *Identificador* { propiedades y métodos }

Identificador: sustantivos, UpperCamelCase

Modificadores: accesibilidad que tendrán sobre ella otras clases

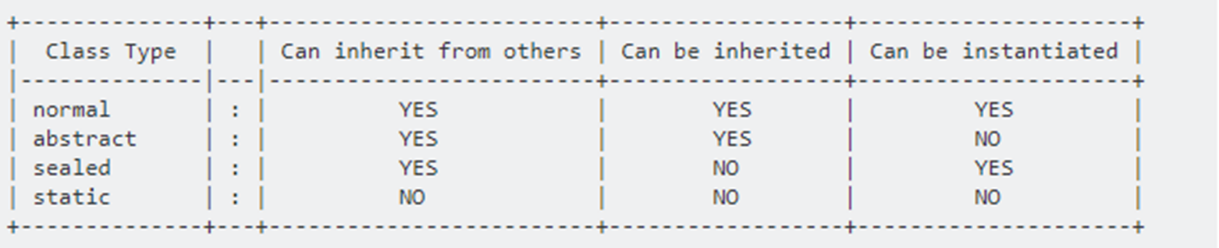
-abstract: la clase no podrá instanciarse. Superclases, estructura base para otras clases, base de jerarquía de clases

-sealed: no heredable (si puede heredar)

-static: No hereda, no heredable, no instanciable

-internal (\*): accesible en todo el proyecto

-public (\*): acc desde cualquier proyecto

-private (\*): accesor por defecto

2.1.2. **Atributos**:

[*modificador*] *tipo* *identificador*;

Identificador: lowerCamelCase. Si es privado \_

Modificador: accedibilidad

-private (\*) miembros de la misma clase. Accesor por defecto

-protected miembros de la misma clase y clases derivadas o hijas.

-internal miembros del mismo proyecto.

-internal protected miembros del mismo proyecto o clases derivadas.

-public Cualquier miembro. Accesibilidad abierta.

-static Indica que es una propiedad de clase.

2.1.3. **Métodos**:

[*modificador*] *retorno* *Identificador* ( [param] ) { Sentencias }

Identificador: verbos, UpperCamelCase

Modificador:

-extern Firma del método (para métodos externos).

-internal (\*) Accesible desde el mismo proyecto.

-public (\*) Accesible desde cualquier proyecto.

-private (\*) Sólo accesible desde la clase.

-protected (\*) Sólo accesible desde la clase o derivadas.

-abstract Sólo la firma del método, sin implementar. Solo en clases abstractas.

-virtual Permite definir métodos, con su implementación, que podrán ser sobrescritos en clases derivadas.

-override Reemplaza la implementación del mismo método declarado como virtual en una clase padre.

-static Indica que es un método de clase.

2.1.3.1. PARÁMETROS OPCIONALES

Por defecto, los parámetros de un método son requeridos. Se puede asignar valores por defecto a los parámetros de un método, haciéndolos opcionales. Cuando se llama al método, si ningún argumento fue suministrado se utilizará el valor por defecto.

método (tipo paramA, tipo paramB=valor) { }

2.1.3.2. METODOS y PROP ESTATICOS (DE CLASE)

Atributos Estáticos

Son atributos o propiedades asociadas a la clase y no a una instancia de la misma. No se necesita instanciar un objeto para acceder a estos atributos. No pueden acceder a los atributos no-estáticos, los cuales son específicos de cada instancia. Se declaran utilizando la palabra reservada static. Se llaman utilizando el nombre de la clase + punto + nombre del atributo o propiedad. Son constantes.

Métodos Estáticos

Son operaciones asociadas a la clase y no a una instancia de la misma. Son lo más parecido a las funciones de los lenguajes estructurados. Se utilizan para procesar datos de entrada y retornar un resultado sin necesidad de acceder al estado / atributos de un objeto concreto. No se necesita instanciar un objeto para llamar a estos métodos. No pueden acceder a los atributos no-estáticos, los cuales son específicos de cada instancia. Se declaran utilizando la palabra reservada static. Se llaman utilizando el nombre de la clase + punto + nombre del método. Solo se pueden llamar desde otros métodos estáticos.

2.1.4. **Constructor:**

modificador Clase (param) { this.propiedad= valorInicial; }

Para inicializar las propiedades al instanciar.

• Constructores por Defecto: Inicializa todos los campos con el valor predeterminado según su tipo.

• Constructores de instancia: inicializan objetos (atributos NO estáticos).

• Constructores estáticos: inicializan clases (atributos estáticos). Utilizan la palabra reservada static. No pueden recibir parámetros.

modificador Clase (tipo paramA, tipo paramB) : this (paramA)

{ this.propiedad= paramB; }

Cuando tengo otro constructor que comparte uno o más parámetros, lo reutilizo. El constructor que no recibe parámetros reutiliza a aquellos que si los reciben, pasando como argumento los valores deseados.

2.1.5. **Destructor**:

~Clase() { liberar memoria y cerrar ficheros }

No se suele usar, la memoria de los objetos la libera el Garbage Collector cuando se pierde la referencia.

2.1.6. **NameSpace**:

namespace *Identificador* { Miembros }

Es una agrupación lógica de clases y otros elementos. Toda clase está dentro de un NameSpace. Proporcionan un marco de trabajo jerárquico sobre el cuál se construye y organiza todo el código. Su función principal es la organización del código para reducir los conflictos entre nombres. Esto hace posible utilizar en un mismo programa componentes de distinta procedencia.

using nameSpace

2.1.7. **Directivas**:

Permiten a un programa identificar los NameSpaces que se usarán en el mismo. Permiten el uso de los miembros de un namespace.

* Using: Permite la especificación de una llamada a un método sin el uso obligatorio de un nombre completamente cualificado.
* Alias: Permite utilizar un nombre distinto para un Namespace. Generalmente se utiliza para abreviar nombres largos.

2.1.8. **Objetos**

*Clase* *identificador* = New *Clase* (*param*);

Instancia de clase. Es un tipo por referencia.

new: asigna memoria.

*Clase* (*param*): constructor para inicializar un objeto en esa memoria

2.1.9. **Interface**:

*modif* interface *identificador* { *retorno* *método* (*claseAbstracta* *param*) }

2.1.10. **Enumerados**

Son un conjunto propio de constantes con nombre. Estos tipos de datos permiten declarar un conjunto de nombres u otros valores literales que definen todos los valores posibles que se pueden asignar a una variable. Por dentro, estas constantes están asociadas con el tipo de dato int.

Normalmente es mejor definir un enum directamente dentro de un espacio de nombres para que todas las clases del espacio de nombres puedan acceder a él con igual comodidad. Sin embargo, un enum también se puede anidar dentro de una clase o struct.

Las enumeraciones, al igual que el resto de las constantes, son propiedades estáticas, de la clase.

modif **enum** identificador { nombreA, nombreN } si no se indican los valores, por defecto arranca en 0 y aumenta en 1

modif **enum** identificador { nombreA, nombreB=valor B, nombreN } puedo dar valor a alguno y los siguientes van en aumento

clase.identificador.nombreA return valorA

2.1.11. **Indexadores**

Permiten a la instancia de una clase ser indexada tal cómo un array. No es necesario indexar con enteros.

private tipo[] atributo = new tipo[N]; // Declaro un array

public tipo **this**[int i] // Defino el indexador

{

**get** { **return** atributo [i]; }

**set** { atributo[i] = **value**; }

}

2.2. Pilares

**2.2.1. Abstracción**:

**Clases abstractas**:

El modificador **abstract** se utiliza para indicar que una clase está incompleta y que sólo se va a utilizar como una clase base. No puede instanciarse. Las clases abstractas se sitúan en la cima de la jerarquía de clases.

Aunque es posible tener variables y valores cuyos tipos en tiempo de compilación sean abstractos, tales variables y valores serán null o contendrán referencias a instancias de clases no abstractas derivadas de los tipos abstractos.

Se permite que una clase abstracta contenga miembros abstractos, aunque no es necesario.

Cuando una clase no abstracta se deriva de una clase abstracta, la clase no abstracta debe incluir implementaciones reales de todos los miembros abstractos heredados; por lo tanto, reemplaza a estos miembros abstractos con la palabra **override**. Una clase derivada de la clase abstracta sí se puede instanciar.

No se puede sellar una clase abstracta.

Las declaraciones de **métodos abstractos** sólo se permiten en clases abstractas. Aunque, por definición formal, un método abstracto es también implícitamente un método virtual, no puede tener el modificador virtual. Una declaración de método abstracto introduce un nuevo método virtual, pero no proporciona una implementación del método, solo la firma.

Por esto es necesario que las clases derivadas no abstractas proporcionen su propia implementación mediante el reemplazo del método. Son métodos y propiedades que se declaran sin implementación.

modif **abstract** class ClaseAbstracta

{ modif **abstract** retorno Metodo (param);

}

modif class ClaseDerivada : ClaseAbstracta

{ modif **override** retorno Metodo (param){ implementacion }

}

**2.2.2. Encapsulamiento**:

Ocultamiento del estado, es decir, de los datos miembro de un objeto de manera que solo se pueda cambiar mediante las operaciones definidas para ese objeto. Se encarga de mantener ocultos los procesos internos.

2.2.2.1. Propiedades:

Miembro que proporciona un mecanismo flexible para leer, escribir o calcular el valor de un campo (privado). Métodos especiales denominados descriptores de acceso (gettes+setters ) / refactorizar

modif tipoDato nombreProp

{ **get** { **return** atributo; }

**set** { **this**.atributo = **value**; }

}

En el Main se usan como atributos en lugar de métodos: obj.Prop = valor; > set

var = obj.Prop; > get

En C:

public void SetProp (tipo valor) { prop = valor; }

public tipoProp GetProp ( ) { return this.prop; }

Atajos:

prop + tab tab > para crear propiedad + accesores

ctrl+r+e > set+get / refactorizar

**2.2.3. Herencia**:

*modif* class *ClaseDerivada* **:** *ClaseBase* { prop+métodos nuevos}

Definir una clase a partir de otra. La clase hija/derivada hereda las propiedades y métodos (algunos o todos) de la clase padre/base, todo excepto los constructores y finalizadores. Jerarquía de clases. Una clase derivada no puede ser más accesible que su clase base. Los miembros private no son visibles en las clases derivadas, pero si se heredan.

Tipos:

Herencia Simple: Una clase derivada puede heredar sólo de una clase base (los lenguajes .NET soportan este tipo de herencia)

Herencia Múltiple: Una clase derivada puede heredar de una o más clases base (C++ soporta este tipo de herencia).

2.2.3.1. Constructores: Base

modif *ClaseDerivada*(param) **:** **base** (param) { sentencias adicionales }

Para llamar a un constructor de la clase base desde un constructor de la clase derivada se usa la palabra reservada base. Si la clase derivada no hace una llamada explícita a un constructor de la clase base, el compilador de C# usará implícitamente un constructor de la forma :base() (constructor sin parámetros de la clase base). Cuando instancio un obj de una clase derivada, antes de llamar a su constructor se llama a los constr de las clases bases, de la más general a la más específica.

2.2.3.2. New: redefinir métodos

Se puede redefinir algo que ya existe en la clase base con la palabra reservada new.

modif **new** retorno MetodoBase () {}

2.2.3.3. Base: ampliar métodos

No reemplaza al método, lo amplía

modif **new** retorno MetodoBase()

{ **base**. MetodoBase(); > llamo al método de la clase base

nuevas instancias que se agregan al método anterior

}

2.2.3.4. Virtual + override

-virtual Permite definir métodos, con su implementación, que podrán ser sobrescritos en clases derivadas.

-override Reemplaza o amplía en una clase derivada la implementación del mismo **método, propiedad, indexador o evento** declarado como **virtual, abstract u override** en una clase padre. El método base reemplazado debe tener la misma firma que el método override (modif de acceso, tipo y nombre).

El método invalidado por una declaración override se conoce como método base invalidado. No se puede reemplazar un método estático o no virtual.

No se pueden usar los modificadores new, static o virtual para modificar un método override.

Determinante de la implementación del método real a invocar:

método virtual> el tipo en tiempo de ejecución de la instancia para la que tiene lugar la invocación

método no virtual> tipo en tiempo de compilación de la instancia

modif class ClaseBase

{ modif **virtual** retorno Metodo (param){ }

}

modif class ClaseDerivada : ClaseBase

{ modif **override** retorno Metodo (param){ }

}

**2.2.4. Polimorfismo**:

Polimorfismo: funciones que tienen el mismo nombre, pero distintos parámetros y quizá se comporten de la misma forma.

Sobrecarga: funciones que se llaman igual, reciben el mismo número de parámetros, pero se aplican a objetos distintos.

objA.Metodo /= objB.Metodo

Clases cuyas propiedades son otra clase.

Cuando creo un método que recibe una clase abstracta que tmb lo puedo usar con las clases derivadas.

3. SOBRECARGA

3.1. Métodos y Constructores

Dos o más métodos en una clase pueden compartir el mismo nombre. Se sobrecargan cambiando el número, el tipo, el orden de los parámetros, modificador de parámetro (out o ref). Los constructores se sobrecargan para instanciar objetos de diferentes formas.

En los métodos reutilizo los métodos con más parámetros dentro de los que tienen menos. En los constructores los que tienen más parámetros utilizan a los que tienen menos con el operador this.

3.2. Operadores

[*acceso*] static *TipoRetorno* operator *Operador* (tipo paramA, tipo paramB) { }

*varTipoRetorno* = paramA operador paramB;

Consiste en modificar su comportamiento cuando este se utiliza con una determinada clase. El modificador de acceso no podrá ser de un ámbito mayor que el de la clase.

Operadores Sobrecargables:

+, -, !, ~, ++, - -, true, false > Unarios

+, -, \*, /, %, &, |, ^, <<, >> > Binarios

==, !=, <, >, <=, >= > Comparación, se deben sobrecargar en pares

No Sobrecargables:

&&, || > Condicionales Lógicos.

[] > Indexador de Array. Se pueden definir indexadores.

() > Casting. Se pueden definir nuevos operadores de conversión.

+=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>= > Asignación. El operador +=, por ejemplo, es evaluado usando el operador +, el cual puede ser sobrecargado.

=, ., ?:, ->, new, is, sizeof, typeof > Estos operadores no se pueden sobrecargar.

3.3. Conversión / casting

Implicitos

[acceso] static implicit operator *nombreTipo*(*tipo* *param*) { }

Explícitos

[acceso] static explicit operator *nombreTipo*(*tipo* *param*) { }

4. FORMS

Programación orientada a eventos. Estilo de programación, no paradigma.

Ante una acción se invoca un método.

Partial Class: permite separar el código de una clase en dos archivos fuentes diferentes. El diseñador de formularios utiliza esta técnica para escribir en un archivo aparte todo el código que él mismo genera.

5. Variables No Escalares

5.1. Arrays

Un array puede ser unidimensional, multidimensional o anidado (jagged).

El valor por defecto es el del tipo de dato.

Los Arrays son reference type, heredan de la clase abstracta System.Array. Implementan la interfaz IEnumerable por lo tanto se pueden iterar usando foreach.

[acceso] tipo[] identificador = new tipo[TAMAÑO]; > unidimensional

tipo[] identificador = {valor1, valorN}; > declarar e inicializar

tipo identificador [A][B]; > declara A arrays de B elementos

[acceso] tipo[ , ] identificador = new tipo[FILAS, COLUMNAS]; > multidimensional (matriz)

tipo[,] identificador = {{valor1}, {valorN}}; > declarar e inicializar

tipo[ ][ ] identificador = new tipo[A][ ]; > array de arrays: puede tener elementos de distinto tamaño

identificador [A] = new tipo [B]; > p/c/ A indico el tamaño de B

5.1.1. Array/matriz de objetos

clase[ ] identificador = new clase [N]; > reservo memoria para el array

for(byte i=0; i<N; i++) { identificador [i] = new clase (); } > reservo memoria para el objeto

Puedo declarar un array de clase base y objetos de clases derivadas.

5.2. Estructuras o registros

Agrupación de datos, llamados campos, no necesariamente del mismo tipo. Se accede a los campos con ‘.’

struct

tipo identificador; > p/declarar una variable del tipo de la estructura

6. Colecciones

Una colección es una clase, de modo que antes de poder agregar elementos a una nueva colección, debe declararla.

6.1. Genéricas

TipoColección<tipoElementos> identificador = new TipoColección < tipoElementos >();

Clases en el NameSpace System.Collections.Generic

Es útil cuando todos los elementos de la colección tienen el mismo tipo de datos. Permiten agregar sólo el tipo de datos deseado.

Dictionary: Representa una colección de pares de clave y valor que se organizan por claves. El tipo de elemento es KeyValuePair<TKey,TValue>

List: Representa una lista de objetos que pueden ser obtenidos mediante un índice. Proporciona métodos para buscar, ordenar y modificar listas.

Queue (FIFO): Representa una colección de objetos con el orden primero en entrar, primero en salir.

Stack (FILO): Representa una colección de objetos con el orden último en entrar, primero en salir.

SortedList: Representa una colección de pares de clave y valor que se ordenan por claves según la implementación de la interfaz IComparer<T> asociada.

6.2. No genéricas

TipoColección identificador = new TipoColección();

Incluidas en el NameSpace System.Collections.

No almacenan los elementos como objetos de un tipo específico, sino como objetos de tipo Object.

ArrayList: Representa una matriz de objetos cuyo tamaño aumenta dinámicamente según sea necesario.

Hashtable: Representa una colección de pares de clave y valor que se organizan por código hash de la clave.

Queue: Representa una colección de objetos con el orden primero en entrar, primero en salir (FIFO).

Stack: Representa una colección de objetos con el orden último en entrar, primero en salir (LIFO).

6.3. Concurrentes

System.Collections.Concurrent

Proporcionan operaciones eficaces y seguras para subprocesos con el fin de obtener acceso a los elementos de colección desde varios subprocesos (hilos).

Deben utilizarse en lugar de sus equivalentes en los espacios de nombres System.Collections.Generic y System.Collections cuando varios subprocesos tienen acceso a la colección simultáneamente.

BlockingCollection<T>on BlockingCollection<T>

ConcurrentDictionary<TKey, TValue>on BlockingCollection<T>

ConcurrentDictionary<TKey, TValue>

ConcurrentQueue<T>

ConcurrentStack<T>.

Casi todas las estructuras dinámicas contienen el método GetEnumerator que devuelve un enumerador para recorrer todos los elementos.

11. SOBRECARGA DE MÉTODOS, POLIMORFISMO, ABSTRACT y VIRTUAL

2do parcial:

15. Excepciones

try { } catch ( ) { }

try: los casos que puedan ser problemáticos

catch: msj de error y/o acciones correctoras

clases de excepciones:

Exception

PathTooLongException

IOException

10. TEST UNITARIOS

11. GENERICS

12. INTERFACES

13. ARCHIVOS DE TEXTO

14. SERIALIZACIÓN

15. BASE DE DATOS

16. THREADS

17. EVENTOS

try{ } catch() { }

18. MÉTODOS DE EXTENSIÓN

|  |  |  |  |
| --- | --- | --- | --- |
| Tema | PPT 2018/campus | PPT clase | Biblio |
| 1. Conceptos Basicos | Ok | ok | ok |
| POO/ | ok | ok | Ok |
| 2. Objetos | ok | ok | ok |
| 3. SOBRECARGA DE OPERADORES | ok | ok | ok |
| 4. FORMS | Ok | igual | - |
| 5. Arrays y | Ok arrays | igual | ok |
| 7 Colécciones | 7 ok | igual | 7 ok |
| 8. ENCAPSULAMIENTO | Prop, index,enum ok | Igual + abstracción ok | Clase 8 ok |
| 9. HERENCIA | ok |  | ok |
| 10., ABSTRACT y VIRTUAL | Abstracción + virtual |  | Virtual+override |
| 11 SOBRECARGA DE MÉTODOS(override), POLIMORFISMO |  |  | ok |
| 15. Excepciones |  |  | ok |
| 16. TEST UNITARIOS |  |  |  |
| 11. GENERICS |  |  |  |
| 12. INTERFACES |  |  |  |
| 13. ARCHIVOS DE TEXTO |  |  |  |
| 14. SERIALIZACIÓN |  |  |  |
| 15. BASE DE DATOS |  |  |  |
| 16. THREADS |  |  |  |
| 17. EVENTOS |  |  |  |
| 18. MÉTODOS DE EXTENSIÓN |  |  |  |

Imprimir ppts 2018 y clase, compararlos