📚 Herència en Programació Orientada a Objectes (POO)

Què és la herència?

La **herència** és un concepte fonamental en la programació orientada a objectes que ens permet **crear una nova classe** (anomenada **subclasse** o **classe derivada**) **a partir d'una altra classe existent** (anomenada **classe base** o **superclasse**).

La *subclasse*:

- **Hereta** totes les **propietats** (atributs) i **mètodes públics i protegits** de la classe base.
- Pot **afegir** nous atributs i mètodes. (Especialització)
- Pot també **modificar** (sobreescriure) comportaments heretats. (Polimorfisme)

🔥 Per què utilitzar la herència?

- **Reutilitzar codi**: Evitem escriure el mateix codi moltes vegades.
- **Crear especialitzacions**: Podem partir d'una estructura comuna i afegir detalls específics.
- **Organitzar millor el codi**: Podem construir **jerarquies** de classes de manera lògica i ordenada.

🎯 Exemple

Suposem que tenim una classe base:

```
class Animal
{
    public string Name {get; set;}
    public int Age {get; set;}
    public string Specie {get; set;}

    public void Eat()
    {
        Console.WriteLine("Estic menjant.");
    }
    public void Sleep()
    {
        Console.WriteLine("Estic dormint.");
    }
}
```

Ara podem crear una subclasse que **hereta** d'Animal:

```
class Dog : Animal
{
    public string Breed {get; set;}

    public void Bark()
    {
        Console.WriteLine("Bup bup!");
    }
}
```

Què passa aquí?

- Gos **ja té** Nom, Edat i Menjar() perquè ho ha heretat d'Animal.
- A més, Gos **afegeix** el seu propi mètode Bordar().

Jerarquies d'herència

L'herència permet crear **jerarquia de classes**.

Exemple:

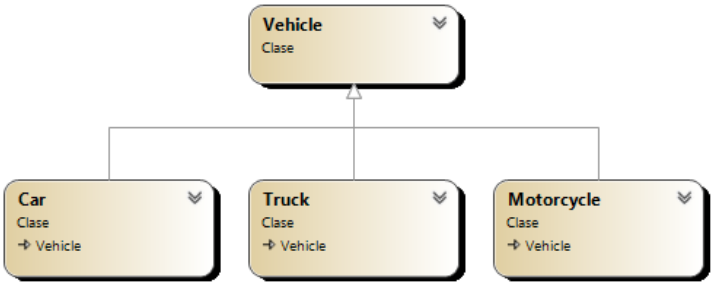
Classe base: Vehicle

|

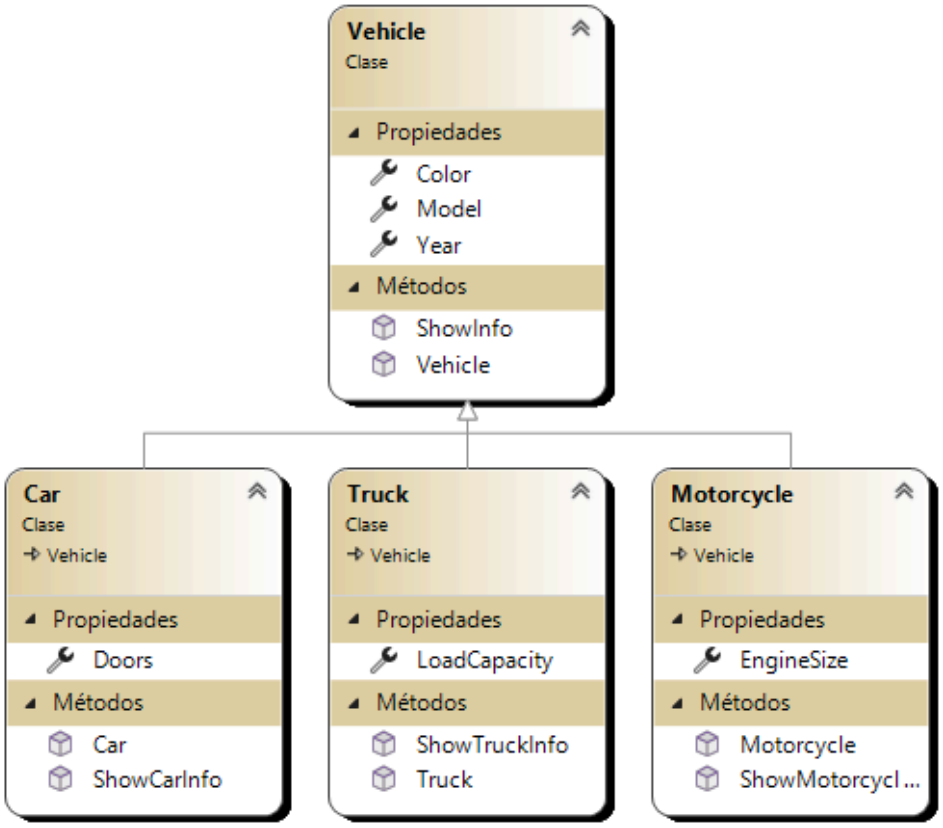
|--> Subclasse: Cotxe

|--> Subclasse: Camió

|--> Subclasse: Motocicleta



Cada subclasse comparteix característiques comuns (per exemple, Color, Any, Model), però afegeix les seves pròpies propietats i comportaments.



🛡️ Accessibilitat i Herència

Quan heretem, els **modificadors d'accés** són molt importants: Els **modificadors d'accés** en C# s'utilitzen per controlar la **visibilitat** i **accessibilitat** dels membres d'una classe (camps, propietats, mètodes i esdeveniments) des de fora de la classe.

Hi ha **quatre modificadors d'accés** disponibles en C#:

1. **Public**:
   Els membres públics són accessibles des de qualsevol lloc, tant **dins** del mateix projecte (**assembly**) com **fora** d'ell.
   Per exemple, un mètode `public` es pot cridar des de qualsevol altra classe, independentment de si està en el mateix projecte o no.

2. **Internal**:
   Els membres `internal` només són accessibles **dins del mateix projecte** (**assembly**).
   Això vol dir que **no** es poden accedir des de fora del projecte, encara que estiguin al mateix espai de noms (**namespace**).
   `Internal` és el **modificador per defecte** per als membres d'una classe si no s'especifica cap modificador.

3. **Private**:
   Els membres `private` només són accessibles **des de dins de la mateixa classe**.
   Això implica que **no** es poden accedir des de fora, encara que sigui dins del mateix espai de noms o projecte.
   Els membres privats es poden considerar com a membres "amagats" accessibles només per la pròpia classe.

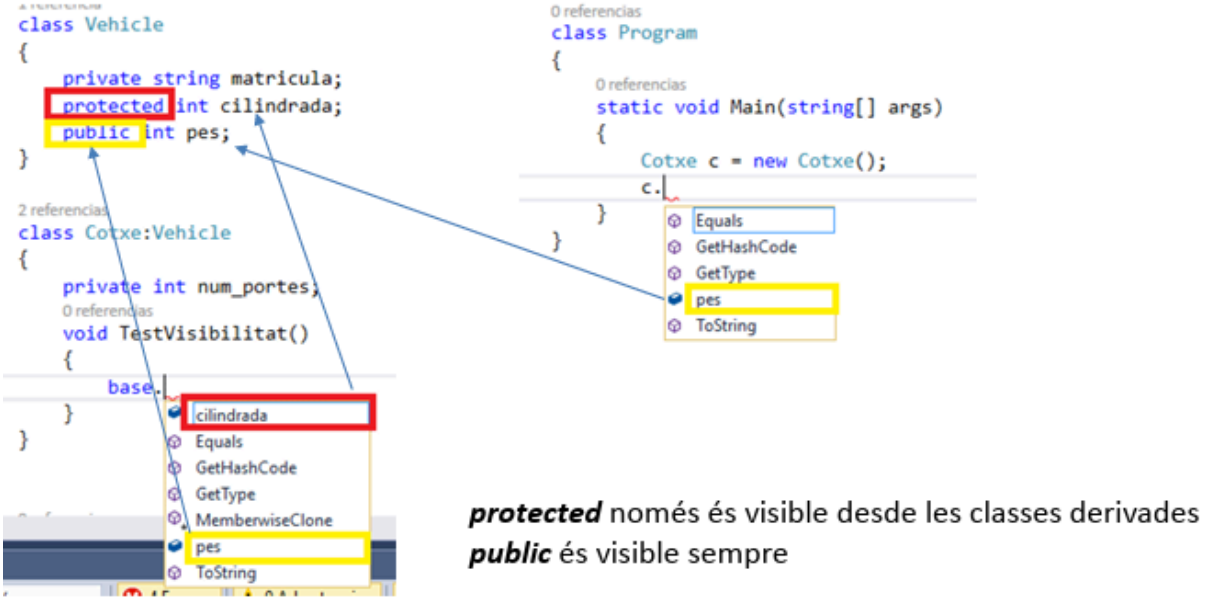
4. **Protected**:
   Els membres `protected` són accessibles **des de la mateixa classe** i **des de qualsevol subclasse** (classe derivada).
   Això vol dir que **no** es poden accedir des de fora de la classe o de les seves subclasses.
   Es poden considerar com a membres private, accessibles només desde les subclasses del codi.

Modificador	Accessible des de la subclasse?	Explicació
public	✅ Sí	Totalment accessible.
private	❌ No	No es pot accedir directament des de la subclasse.
protected	✅ Sí	Només accessible dins de la subclasse o la pròpia classe.
internal	✅ Només si és dins el mateix projecte	Accessible si està en el mateix "assembly".

🚀 Conclusió:

- Herència = **Reutilitzar + Especialitzar + Organitzar**.
- Una subclasse **és un tipus** de la classe base ("un gos **és un** animal").
- Hem de tenir cura amb **quins membres** exposem i **com** els gestionem.

Exemple amb una classe Vehicle amb 3 atributs, un privat, un protegit i un públic...



protected només és visible desde les classes derivades
public és visible sempre

🛠️ Com heretar i implementar interfícies en C#

En C#, una **classe** pot:

- **Heretar** d'una **classe base** (utilitzant : i el nom de la classe base).
- **Implementar** una o més **interfícies** (també amb :, posant després de la classe base els noms de les interfícies).

```
class Dog : Animal
```

Important:

- **Només podem heretar** d'**una sola classe base** (C# no permet herència múltiple de classes!).
- **Podem implementar tantes interfícies com vulguem**.

```
class Subclasse : ClasseBase, IInterficie1, IInterficie2
{
    // Implementació dels mètodes de l'interfície
    // Més codi propi de la classe
}
```

🎯 Exemple

```
public class Figure
{
	protected string name;
	protected int borderSize;

	public Figure(string name, int borderSize)
	{
		this.name = name;
		this.borderSize = borderSize;
	}
	public void ResetBorder()
	{
		borderSize = 0;
	}
	public override String ToString()
	{
		return "NAME = " +name+ " BORDER SIZE = " +borderSize+ "\n";
	}
}
```

```
public class Circle : Figure
{
	private double radius;
	private Point center; //Point class solved in previous handouts.

	public Circle(Point p, double r, string n, int bS) : base(n, bS)
	{
		this.center = p;
		this.radius = r;
	}
	public void Translate(double x, double y)
	{
		this.center.x = this.center.x + x;
		this.center.y = this.center.y + y;
	}
	public override string ToString()
	{
		StringBuilder info = new StringBuilder();
		info.AppendLine(base.ToString());
		info.AppendLine("CENTER = " + center.ToString());
		info.AppendLine("RADIUS = " + this.radius);
		return info.ToString();
	}
}
```

```
public class Square
{
	private double sideLength;

	public Square(string n, int bS, double side) : base(n, bS)
	{
		this.sideLength = side;
	}
	// EXERCISE : Write the ToString method!
}
```

Exercici: Ús de classes → Indica quines frases són correctes i quines són incorrectes, i dóna una explicació.

Si la frase és correcta, indica quin mètode específic s'executa.

```
Figure sq = new Square("ONE SQUARE", 2 , 10.6);
Point p = new Point(0,0);
Figure c = new Circle(p, 4.3, "ONE CIRCLE", 20);
Circle c2 = new Circle(p, 2.5, "ANOTHER CIRCLE", 30);
Console.WriteLine(sq);
Console.WriteLine(c);
Console.WriteLine(c2);
c.Translate(5,5);
c2.Translate(5,5);
c.ResetBorder();
c2.ResetBorder();
```

🧩 CLASSES ABSTRACTES

Una **classe abstracta** és una classe que **no es pot instanciar directament** (no es pot crear un objecte d'aquesta classe), però que **pot ser heretada** per altres classes.

Per a ser una classe abstracta ha de tenir **com a mínim un mètode abstracte**, un mètode abstracte consisteix en declarar el mètodes sense implementar-lo, obligant a les classes derivades a proporcionar una implementació sobre aquests mètodes. De fet, el significat d'un mètode abstracte radica en el fet que a la clase base ens falta informació per poder-lo implementar, en canvi, a les subclasses derivades sí que el podem implementar.

L'objectiu d'una classe abstracta és **proporcionar una definició comuna** d'una classe base perquè múltiples subclasses la puguin heretar.

Les classes abstractes sovint s'utilitzen per **definir un conjunt de funcionalitats** que les subclasses derivades **han d'implementar**. Aquesta característica també la tenien les interfícies.

Una classe abstracta pot tenir **tant membres abstractes** com **no abstractes** (mètodes i propietats).

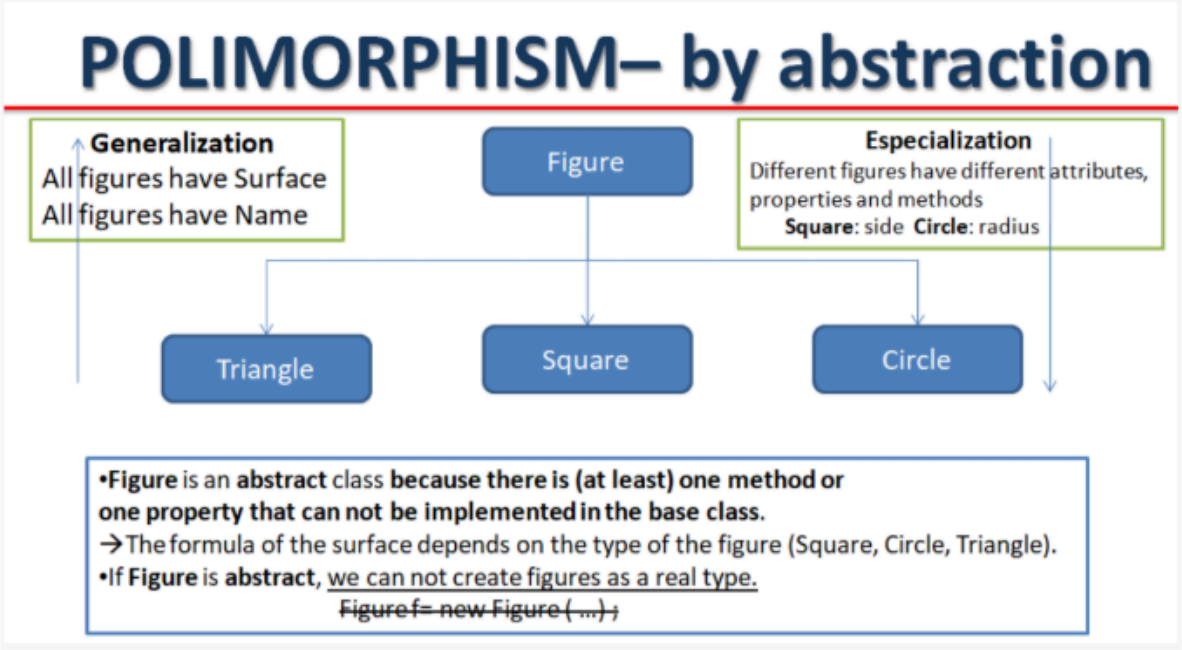
- Un **membre abstracte** és aquell que es **declara** però **no es defineix** a la classe abstracta.
  És a dir, la **implementació** d'aquest membre **l'ha de proporcionar** qualsevol classe que hereti d'aquesta classe base.

- Els **membres no abstractes**, en canvi, **tenen una implementació** definida a la classe abstracta i poden ser heretats directament per qualsevol subclasse.

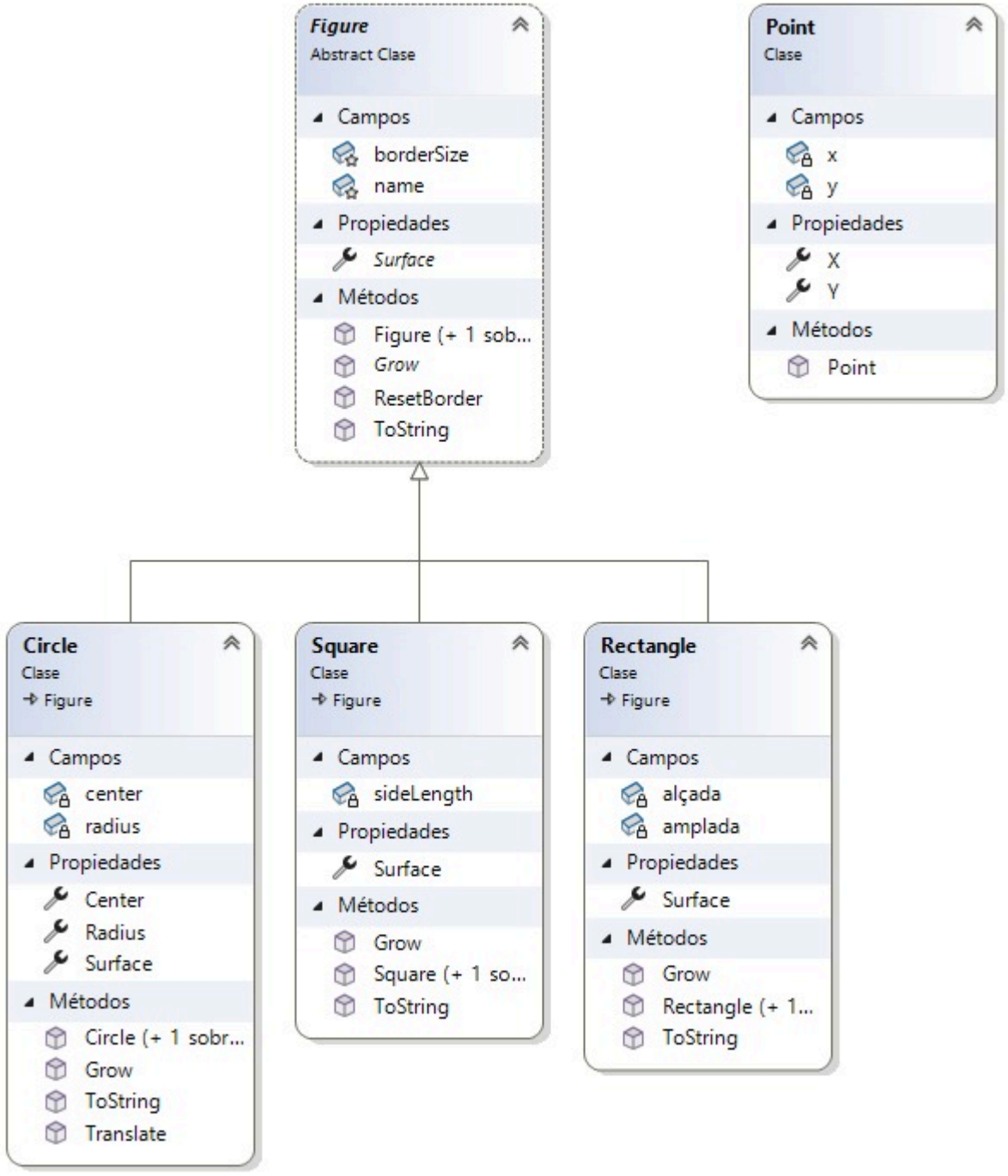
Una classe abstracta **ha de tenir almenys un mètode o propietat** que **no es pugui implementar** dins la pròpia classe (abstracte) i que **s'hagi d'implementar** obligatòriament a les subclasses que l'hereten.
Dit d'una altra manera, **si qualsevol membre d'una classe és abstracte, la classe es converteix automàticament en abstracta**.

🎯 Exemple pràctic:

En l'exemple següent, **no podem calcular la superfície** només tenint una `Figure`, però sí que podem calcular-la a les classes derivades.
Per aquest motiu, `Surface` és una propietat abstracta.



Declaració d'una classe abstracta en C#



Per declarar una classe abstracta, cal escriure el modificador `abstract` **davant de la classe**.

Qualsevol mètode o propietat abstracte també ha d'anar marcat amb `abstract`, i només es declara la signatura (acaba amb punt i coma), igual que a les interfícies.

```
public abstract class Figure
{
    protected string name;
    protected int borderSize;

    public Figure(string name, int borderSize)
    {
        this.name = name;
        this.borderSize = borderSize;
    }

    public abstract double Surface { get; }  // Propietat abstracta
    public abstract void Grow(double increment); // Metode abstracte

    public void ResetBorder()
    {
        borderSize = 0;
    }

    public override string ToString()
    {
        return "NOM = " + name + " MIDA DE LA VORA = " + borderSize;
    }
}
```

- En aquest exemple, tenim:
  - `Surface` (propietat abstracta)
  - `Grow (double increment)` (mètode abstracte)

🧩 Implementació de mètodes abstractes en subclasses

Per implementar els mètodes abstractes a les subclasses, utilitzem el modificador `override`:

Exemple amb la classe `Circle`:

```
public class Circle : Figure
{
    private double radius;
    private Point center;

    public Circle(Point p, string n, int bS, double radius) : base(n, bS)
    {
        this.center = p;
        this.radius = radius;
    }
    // Implementem Propietat abstracta heretada de Figura
    public override double Surface
    {
        get { return Math.PI * radius * radius; }
    }
    // Implementem el mètode abstracte heretat de Figure
    public override void Grow(double increment)
    {
        this.radius += increment;
    }
}
```

🎯 Explicació addicional:

- **La superfície d'un quadrat** seria: costat × costat.
- **La superfície d'un cercle** seria: π × radi².
- **Fer créixer un quadrat** implica augmentar la mida dels costats.
- **Fer créixer un cercle** implica augmentar el radi.

🧪 Exercici proposat:

Implementa els mètodes override de la classe `Square` i **prova** totes les classes amb aquest programa `Main`:

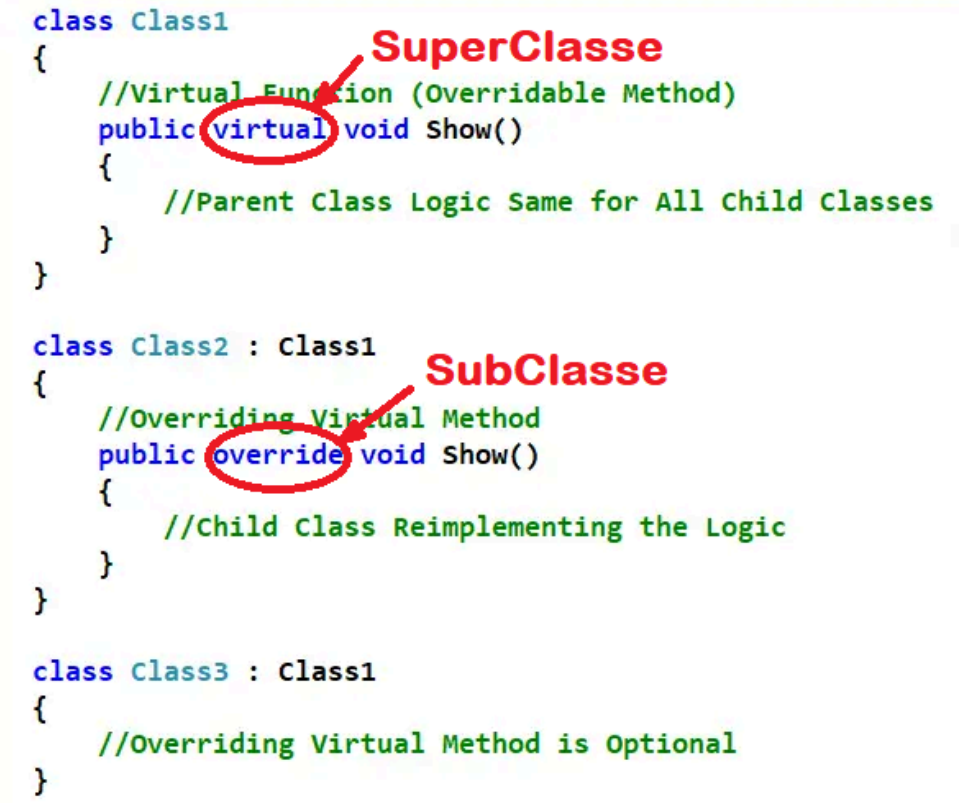
```
static void Main(string[] args)
{
	List<Figure> figures = new List<Figure>();
	Random r = new Random();
	int figureSelected;
	Figure f = null;

	for (int i = 0; i < 20; i++)
	{
		figureSelected = r.Next(2);
		if (figureSelected == 0)
		{
			Point p = new Point(i, i);
			f = new Circle(p, "CIRCLE" + i, r.Next(50), 1);
		}
		else
			f = new Square("SQUARE" + i, i, r.Next(50));

		f.Grow(1000);
		figures.Add(f);
	}
	foreach (Figure unaFigura in figures)
	{
		Console.WriteLine(unaFigura);
		Console.WriteLine("SURFACE: " + unaFigura.Surface);
	}
}
```

Atenció: Fixeu-vos que utilitzem el **tipus aparent** `Figura`, però realment estem treballant amb `Cercle` i `Quadrat`.

🧩 MÈTODES I PROPIETATS VIRTUALS




```
class Class1
{
    //Virtual Function (Overridable Method)
    public virtual void Show()
    {
        //Parent Class Logic Same for All Child Classes
    }
}

class Class2 : Class1
{
    //Overriding Virtual Method
    public override void Show()
    {
        //Child Class Reimplementing the Logic
    }
}

class Class3 : Class1
{
    //Overriding Virtual Method is Optional
}
```

Un **mètode** o **propietat virtual** és un membre d'una classe que pot ser **sobreescrit** (sobre escrit) per una subclasse.
Això vol dir que un membre virtual pot tenir una implementació **diferent** a la subclasse respecte de la classe base.

Ja coneixes alguns mètodes virtuals: `ToString`, `Equals` i `GetHashCode` són tots mètodes virtuals.

Un mètode virtual **pot ser implementat** tant a la classe base com a la classe derivada.

- A la **classe base** cal incloure el modificador `virtual`.
- A la **classe derivada** cal incloure el modificador `override`.

Tant les classes **abstractes** com les **no abstractes** poden tenir mètodes virtuals.

Pots **reutilitzar** el codi de la classe base fent servir la referència `base` (per exemple: `base.ToString()`).

Fins i tot pots cridar el mètode virtual de la base i després escriure codi específic per especialitzar el comportament a la classe derivada que sobrescriu el mètode.

🎯 Exemple

```
public class Animal
{
    public virtual void MakeSound()
    {
        Console.WriteLine("ESTIMO EL MEU AMO");
    }
}

public class Dog : Animal
{
    public override void MakeSound()
    {
        Console.WriteLine("GUAU GUAU!");
    }
}

public class Cat : Animal
{
    public override void MakeSound()
    {
        base.MakeSound();
        Console.WriteLine("MIAU MIAU!");
    }
}
```

EXERCICI

Imagina que cada figura té un **preu fix** que es calcula multiplicant la **mida del contorn** per **1 €**, **excepte** si el **nom** conté la paraula "GOLD".
En aquest cas, s'ha de multiplicar la mida del contorn per **100 €**.

A més a més:

- Un **cercle** té un **augment** en el preu, sumant el **radi** multiplicat per **0,5**.
  Si és un **cercle "GOLD"**, s'ha de multiplicar per **50** en comptes de 0,5.

- Un **quadrat** té un **augment** de **0,5** multiplicat pel **perímetre** del quadrat, i multiplicat per **50** si és un **"GOLD Square"**.

Implementa la propietat **virtual** `Price` a la classe abstracta `Figure` i la **reescrita** `Price` a cada figura específica (`Square` i `Circle`).