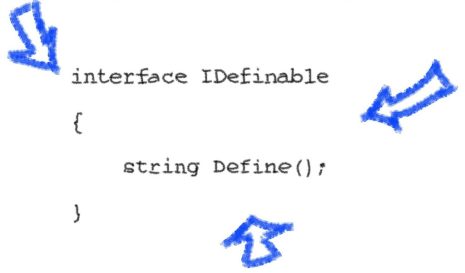
**Rozhraní, dědičnost, abstraktní třídy a přetěžování**

## **Rozhraní**

**Rozhraní** (Interface) je kontrakt, který specifikuje operace, které má třída splňovat, a který se již nezabývá tím, jak toho bude konkrétně dosaženo. Rozhraním objektu se myslí to, jak je objekt viditelný zvenku. Rozhraní objektu tvoří právě jeho veřejné metody, je to způsob, jakým s určitým typem objektu můžeme komunikovat. Výhodou rozhraní je, že pro třídu můžeme implementovat více rozhraní. Při dědění přebírá třída od rodičovské třídy jeho rozhraní. Je zvykem začínat názvy rozhraní velkým **I**.

Příklad: Představte si, že máte ZOO. V ZOO máte různá zvířata, počínajíc slony, končící kolpíky. Veškeré zvěrstvo má sdílenou vlastnost, že dýchá. V takovém případě bychom si vytvořili interface IZvire, jenž by obsahoval sdílenou vlastnost Dychej().

Dědění rozhraní se zapisuje stejnou syntaxi jako dědění tříd a to class Child:IZvire{}



## 

## 

## **Dědičnost**

**Dědičnost** (Inheritance) je spolu se zapouzdřením a polymorfismem jednou ze tří primárních charakteristik objektově orientovaného programování. Dědičnost umožňuje vytvářet nové třídy, které znovu používají, rozšiřují a upravují chování definované v jiných třídách.

* **Parent class** (rodičovská třída)- Třída, od které ostatní třídy dědí
* **Child class** (potomek) - Třída, která dědí od rodičovské třídy

**Child class** může dědit pouze od jedné **Parent class**(toto platí v jazyce Java i C#, některé jazyky ale vícenásobnou dědičnost povolují, např. C++).

Dědičnost je však přechodná. Pokud je **ClassC** odvozena od **ClassB** a **ClassB** je odvozena od ClassA, ClassC zdědí členy deklarované v ClassB a ClassA.

Třídu můžeme nastavit již dále **neděditelnou**.

* Java - klíčové slovo **final,** příklad: public final class Person{ }
* C# - klíčové slovo **sealed**, příklad: public sealed class Person{ }

Stejně je možné nastavit i **metody**.

Proměnné a metody můžeme nastavit jako **protected** a tím zamezit přístup jiným třídám, ale bude možné nadále dědit. V C# třída dědí pomocí znaku “**:**” (stejně jako při implementaci rozhraní), v Javě se používá slovo **extends**.

**Příklad**(C#): class User : Person { }

class Person

{

private string firstName

protected string FirstName{

get {return firstName;}

set {name=value;}

}

protected string LastName { get; set; }

protected byte Age { get; set; }

protected string FullName => $"{FirstName} {LastName}";

protected Person(string firstName, string lastName, byte age)

{

FirstName = firstName;

LastName = lastName;

Age = age;

}

protected string Introduce() => $"Hi, I'm {FullName}.";

}

// Employee.cs

class Employee : Person

{

public string JobTitle { get; set; }

public decimal Salary { get; set; }

public Employee(string firstName, string lastName, byte age, string jobTitle, decimal salary): base(firstName, lastName, age)

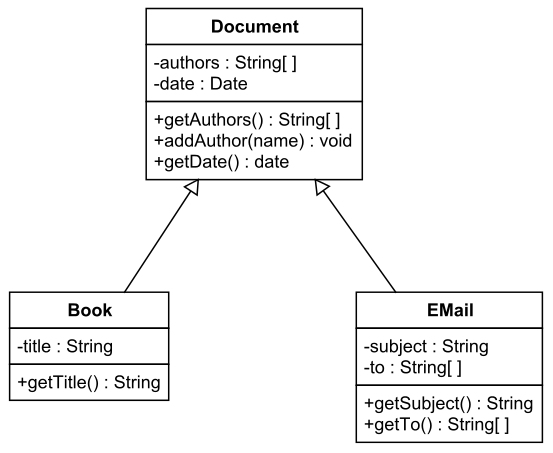
{

JobTitle = jobTitle;

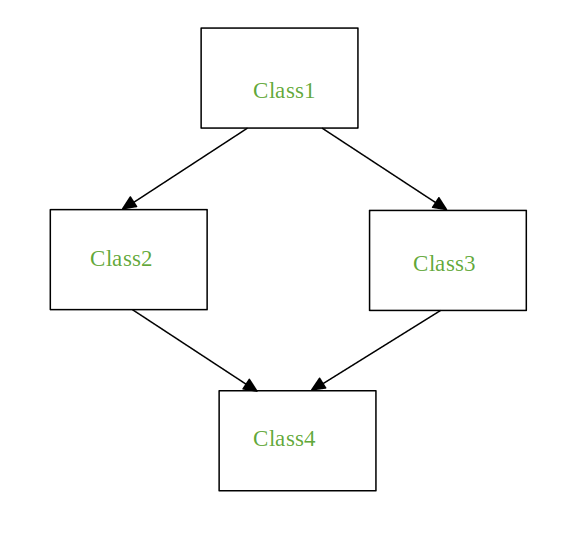
Salary = salary;

}

}



**Multiple Inheritance in python (Deadly diamant problem)**



# Python Program to depict multiple inheritance

# when method is overridden in both classes

**class** Class1:

**def** m(self):

        print("In Class1")

**class** Class2(Class1):

**def** m(self):

        print("In Class2")

**class** Class3(Class1):

**def** m(self):

**print**("In Class3")

**class** Class4(Class2, Class3):

**pass**

obj **=** Class4()

obj.m() 🡪 "In Class2" or "In Class3"

## **Polymorfismus**

Představme si, že máte v ruce mobilní telefon. Za předpokladu, že chcete komunikovat, mobilní telefon je toho schopen mnoha způsoby. Může volat, může psát zprávu, poslat obrázek, mail a tak dále. Účel všech těchto akcí ale zůstává stejný, komunikace. To samé by platilo pro vozidlo s hybridním pohonem, jenž by bylo schopné jet jak na fosilní paliva, tak čistě elektrický pohon. V obou případech jde o akci pohybu, leč jiným způsobem.

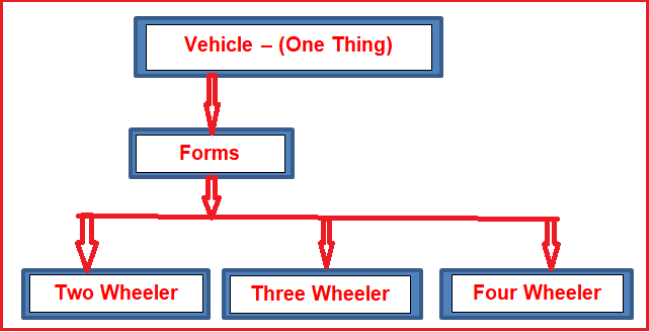
Trošku jiným příkladem je entita vozidlo. Ta může mít 2, 3, 4, 6 a více kol, avšak stále zůstává vozidlem a jeho účelem je pohyb.

Polymorfismus je označení pro stav, kdy v množině existuje pro určitý úkaz minimálně 2 varianty.

Polymorfismus je často označován jako třetí pilíř objektově orientovaného programování po zapouzdření a dědičnosti. Polymorfismus je řecké slovo, které znamená "mnoho-formoval" a má dva odlišné aspekty:

* Za běhu objekty odvozené třídy mohou být považovány za objekty základní třídy v místech, jako jsou parametry metody a kolekce nebo pole. Dojde-li k tomuto polymorfismu, deklarovaný typ objektu již není shodný s typem za běhu.
* Základní třídy mohou definovat a implementovat virtuální *metody a* odvozené třídy je mohou přepsat, což znamená, že poskytují vlastní definici a implementaci. Za běhu, když kód klienta volá metodu, CLR vyhledá typ běhu objektu a vyvolá toto přepsání virtuální metody. Ve zdrojovém kódu můžete volat metodu na základní třídy a způsobit odvozené třídy verze metody, které mají být provedeny.

Virtuální metody umožňují pracovat se skupinami souvisejících objektů jednotným způsobem. Předpokládejme například, že máte kreslicí aplikaci, která umožňuje uživateli vytvářet různé druhy obrazců na kreslicí ploše. V době kompilace nevíte, které konkrétní typy obrazců uživatel vytvoří. Aplikace však musí sledovat všechny různé typy obrazců, které jsou vytvořeny a musí je aktualizovat v reakci na akce myši uživatele. Polymorfismus můžete použít k vyřešení tohoto problému ve dvou základních krocích:

1. Vytvořte hierarchii tříd, ve které každá konkrétní třída tvarů pochází ze společné základní třídy.
   1. Virtuální metoda slouží k vyvolání příslušné metody pro všechny odvozené třídy prostřednictvím jednoho volání metody základní třídy.****

## Method overloading

Jedná se o schopnost mít více metod ve stejné třídě se stejným jménem pouze s jinými vstupními parametry. Každá taková metoda má vlastní implementaci a každá se může chovat úplně jinak.

Známe ji taky jako compile-time (nebo static) polymorphismus protože všechny overloaded metody jsou vyhodnoceny ve chvíli kdy se aplikace kompiluje.

public class Calculator

{

public int AddTwoInt(int a, int b) => a + b;

public int AddThreeInt(int a, int b, int c) => a + b +c;

public double AddTwoDouble(double a, double b) => a + b;

public double AddThreeDouble(double a, double b, double c) => a + b + c;

}

Namísto předchozí ukázky kde každá metoda jiný název lze všechny metody pojmenovávat stejně jen zanechat jiné parametry na vstupu metody.

public class Calculator

{

public int Add(int a, int b) => a + b;

public int Add(int a, int b, int c) => a + b +c;

public double Add(double a, double b) => a + b;

public double Add(double a, double b, double c) => a + b + c;

}

## Method overriding

Je to schopnost přepsat implementaci metody v třídě od děděné od rodiče. Když je metoda overriden, jméno a vstupní i výstupní parametry zůstávají stejný, ale implementace se může změnit (přepsat) následně závisí na tom metoda z jaké třídy (od děděné, rodiče) je volaná.

Známe ji také jako runtime (nebo dynamic) polymorphismus protože typ objektu není předem jistý až za běhu programu. A z toho důvodu jaká z implementací metody se použije je jasné až za běhu programu

public class Dog

{

public virtual void Woof()

{

Console.WriteLine("Woof!");

}

}

public class YappyDog : Dog

{

public override void Woof()

{

Console.WriteLine("Woof! Woof! Woof!");

}

}