**Základy OOP**   
(třída a instance, datové položky, vlastnosti, metody, konstruktory, statické prvky, dědičnost, operátory přístupu, překrývání, abstrakce, přetěžování metod)

**ŘEŠÍME OOP V C#**

**Zapouzdření (Encapsulation):**

* Zapouzdření je mechanismus, který skrývá vnitřní stav objektu a omezuje přístup k jeho vnitřním detailům. To znamená, že údaje jsou ukryty a přístup k nim je umožněn pouze pomocí veřejných metod (metody přístupu), které mohou kontrolovat a validovat přístup k datům.
* Příklad: Máme třídu "Auto" s datovou položkou "rychlost". Přistupujeme k ní pomocí metody "NastavRychlost" místo přímé manipulace s datovou položkou, což umožňuje kontrolovat platnost hodnoty rychlosti.

**Dědičnost (Inheritance):**

* Dědičnost umožňuje vytvářet nové třídy na základě existujících tříd. Podtřída může zdědit vlastnosti a metody nadřazené třídy. Tím se zvyšuje opětovné použití kódu a umožňuje vytvářet hierarchie tříd.
* Příklad: Máme třídu "Vozidlo" a z ní odvozené třídy "Auto" a "Motorka". "Auto" a "Motorka" zdědí vlastnosti a metody třídy "Vozidlo".

**Polymorfismus (Polymorphism):**

* Polymorfismus umožňuje používat stejné jméno pro metody, ale s různým chováním. Může se jednat o přetěžování metod (metoda má stejné jméno, ale různé parametry) nebo o překrývání metod (metoda s tímto samým názvem v nadřazené a podřazené třídě).
* Příklad: Máme třídu "Tvar" s metodou "VypocitejPlochu". Potom můžeme mít podtřídy "Kruh" a "Čtverec", které překrývají tuto metodu pro vlastní výpočet plochy.

**Abstrakce (Abstraction):**

* Abstrakce je proces skrývání podrobností a poskytování pouze nezbytných informací. Abstraktní třídy a rozhraní poskytují obecný pohled na funkcionalitu, ale konkrétní implementace může být poskytnuta až v odvozených třídách.
* Příklad: Máme abstraktní třídu "Zvire" s abstraktní metodou "VydavejZvuk". Tuto třídu mohou implementovat konkrétní zvířata, jako jsou "Pes" nebo "Kočka".

**KECY OKOLO**

# DRY:

* DRY je zkratka pro "Don't Repeat Yourself". Tato zásada zdůrazňuje snahu minimalizovat opakování kódu v projektu. Místo toho, abychom stejný kód psali opakovaně, měli bychom ho extrahovat do jednoho místa a znovu používat toto místo.
* Proč:
  + Snížení chyb: Když měníme kód, který je napsán pouze jednou, nemusíme se starat o to, že zapomeneme aktualizovat všechny kopie kódu.
  + Údržba: Změny v chování můžeme provádět na jednom místě, což zjednodušuje údržbu.

# SOLID:

[**https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/**](https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/)

Principy SOLID je pět zásad objektově orientovaného návrhu tříd. Jedná se o soubor pravidel a osvědčených postupů, které je třeba dodržovat při návrhu struktury tříd.

**SRP - Jednotná odpovědnost (Single Responsibility Principle):**

* Třída by měla mít pouze jednu odpovědnost a měla by mít jen jediný důvod ke změně. To zajišťuje jednotnost a snižuje riziko chyb.

**OCP - Otevřené/zavřené (Open/Closed Principle):**

* Software by měl být otevřen pro rozšíření, ale zavřen pro modifikaci. Měli bychom moci přidat novou funkcionalitu bez změny existujícího kódu.

**LSP - Liskov substituce (Liskov Substitution Principle):**

* Objekty odvozené od dané třídy by měly být schopny být nahrazeny instancemi této třídy bez ovlivnění správnosti programu.

**ISP - Rozhraní segregation (Interface Segregation Principle):**

* Mnoho rozhraní specifických pro klienta je lepší než jedno univerzální rozhraní. Klienti by neměli být nuceni implementovat funkci, kterou nepotřebují.

**DIP - Závislosti inversion (Dependency Inversion Principle):**

* Vyšší úrověň modulů by neměla záviset na nižší úrovni. Závislosti by měly být zaměřeny na abstrakce, ne na konkrétní implementace.

# Proč OOP:

* Lepší čitelnost kódu
* Znovupoužitelnost
* Snazší Debugování:
  + Lepší struktura a organizace kódu v OOP usnadňuje ladění a hledání chyb. Když každá třída má jasně definovanou odpovědnost - může být snazší identifikovat, kde se problémy vyskytují.
* Komentáře a Dokumentace:
  + OOP umožňuje lépe organizovat kód, což může snížit potřebu nadměrných komentářů. Díky jasným třídám a metodám může být kód samo o sobě snadno pochopitelný
* Rozšiřitelnost:
  + Díky principům OOP lze software snadno rozšířit

# Třída a Instance

**Třída:**

* Základ OOP
* Šablona nebo plán pro vytváření objektů
* Je to jakýsi návod, který definuje vlastnosti (atributy) a chování (metody) objektů

**Instance:**

* Konkrétní objekt vytvořený na základě třídy
* Konkrétní realizace nebo exemplář třídy s konkrétními hodnotami atributů

# Datové položky

* Číselné:
  + **Inter types** | range | velikost v **bites** :
    - Byte | 0-255 | 8
    - Sbyte | -128-127 | 8
    - Short | -32 768 – 32 767 | 16
    - ushort | 0 – 65 535 | 16
    - Int | - 2 147 483 648 – 2 147 483 647 | 32
    - uint | 0 – 4 294 967 295 | 32
    - Long | 64-bit integer
    - ulong | 64-bit integer
  + **Floating point types:**
    - Float | pro 6 – 9 míst za čárkou | 32
    - Double | pro 15 - 17 míst za čárkou | 64
    - Decimal | pro 28-29 míst za čárkou | 128
* String – 2bytes per character
* Bool - 1 bit
* Char – 2 bytes

# Vlastnosti (properties)

* Spadají pod pilíř „Encapsulation / Zapouzdření“
* Get and Set
* Prvně se deklarují proměnné jako privátní
* Pomocí public Getteru a Setteru se s proměnnou pracuje

A screen shot of a computer program

Description automatically generated

**Proč?**

* Protože v setteru můžeme validovat data

A computer screen shot of a program

Description automatically generated

Také můžeme nastavit pouze read-only vlasnost:



# Metody

* Metoda je blok kódu, který se spustí pouze tehdy, když je zavolán
* Do metody lze předávat data, tzv. parametry
* Metody slouží k provádění určitých akcí a jsou také známy jako funkce
* Proč? K opakovanému použití kódu: jednou definuji kód a mnohokrát ho použiji

A black screen with white text

Description automatically generated

* **Modifikátor přístupu:** Určuje, kdo má přístup k této metodě (např. public, private, protected)
* **Návratový typ**: Specifikuje typ hodnoty, kterou metoda může vrátit. Pokud metoda nevrací hodnotu, použije se void
* **Název metody:** Prostě název
* **Parametry:** Informace, které metoda může přijmout. Může být nula nebo více parametrů oddělených čárkami
* **Tělo metody:** Co ta metoda má dělat
* **Return:** Volitelný příkaz pro vrácení hodnoty z metody (používá se pouze u metod s návratovým typem jiným než void)

# Konstruktory

* metoda třídy, která je volána při vytváření instance (objektu) této třídy
* Hlavním účelem je inicializovat stav objektu nebo provést jiné operace nezbytné při jeho vytváření. ==
* Mají stejné jméno jako třída a nemají návratový typ
* Mohou mít i parametry
* Mohou být přetíženy

A computer screen shot of a program code

Description automatically generatedA screen shot of a computer code

Description automatically generatedA screen shot of a computer program

Description automatically generated

# Statické prvky

* Jsou takové prvky (členy třídy nebo metody), které jsou přidruženy k třídě, nikoliv k instanci této třídy
* To znamená, že jsou společné pro všechny instance této třídy a nemohou být volány na konkrétní instanci
* Statické prvky jsou přístupné pomocí jména třídy, nikoliv pomocí instance této třídy
* Značí se *static*

A screen shot of a computer code

Description automatically generated

Jak je vidět, není potřeba vytvářet nový objekt pro „Secti“, můžeme ji totiž rovnou zavolat.

# Dědičnost (inheritance)

* **Rodičovská třída (nadřazená třída):**
  + Je to existující třída, ze které je odvozena nová třída. Tato třída poskytuje základní implementaci a funkcionalitu, která může být sdílena s odvozenými třídami.
* **Potomkovská třída (podřazená třída):**
  + Je to nová třída, která je vytvořena na základě existující rodičovské třídy. Potomkovská třída dědí atributy a metody rodičovské třídy, může přidat další funkce nebo upravit existující.

A computer code with text

Description automatically generated with medium confidence

# Operátory přístupu (access modifiers)

A screenshot of a computer

Description automatically generated

Idk co po mne chcete , ta tabulka dava snad smysl

# Překrývání (override)

* Umožňuje potomkovské třídě poskytnout vlastní implementaci metody, kterou zdědila od své rodičovské třídy
* Tedy: potomkovská třída poskytuje novou implementaci metody se stejným názvem, jako je metoda v rodičovské třídě
  + Rodičovská třída poskytuje výchozí implementaci metody, kterou může potomek překrýt
  + Potomek označí metodu klíčovým slovem override, čímž naznačí, že se jedná o překrývání
  + Metoda v potomkovské třídě musí mít stejnou signaturu (název, návratový typ a parametry) jako metoda v rodičovské třídě

A screen shot of a computer program

Description automatically generated

A computer screen with white text and black background

Description automatically generated

# Abstrakce

* Abstraktní třídy:
  + Abstraktní třída je třída, která nemůže být instanciována a slouží pouze jako vzor pro odvození (inheritance) od ní. Obsahuje a definuje abstraktní (neimplementované) metody, které musí být implementovány v potomkovských třídách.
* Abstraktní metody:
  + Abstraktní metoda je metoda v abstraktní třídě, která nemá implementaci v samotné abstraktní třídě. Implementace těchto metod musí být poskytnuta v potomkovských třídách.

A screen shot of a computer code

Description automatically generated

# Přetěžování metod

* Díky přetěžování metod může mít více metod stejný název, ale různé parametry:

A group of black and blue text

Description automatically generated

# Override vs abstract

Virtual methods have an implementation and provide the derived classes with the option of overriding it. Abstract methods do not provide an implementation and force the derived classes to override the method.

So, abstract methods have no actual code in them, and (non-abstract) subclasses HAVE TO override the method. Virtual methods can have code, which is usually a default implementation of something, and any subclasses CAN override the method using the override modifier and provide a custom implementation.

# Interface vs Abstract

* interfaces can have no state or implementation
* a class that implements an interface must provide an implementation of all the methods of that interface
* abstract classes may contain state (data members) and/or implementation (methods)
* abstract classes can be inherited without implementing the abstract methods (though such a derived class is abstract itself)
* interfaces may be multiple-inherited, abstract classes may not (this is probably the key concrete reason for interfaces to exist separately from abtract classes - they permit an implementation of multiple inheritance that removes many of the problems of general MI).

Abstract classes are used for **Modelling** a class hierarchy of similar looking classes (For example Animal can be abstract class and Human , Lion, Tiger can be concrete derived classes)

AND

Interface is used for **Communication** between 2 similar / non similar classes which does not care about type of the class implementing Interface(e.g. Height can be interface property and it can be implemented by Human , Building , Tree. It does not matter if you can eat , you can swim you can die or anything.. it matters only a thing that you need to have Height (implementation in you class) )