Objektové programování II

Zapouzdření, dědičnost, polymorfismus, přepsání a překrytí (virtual, override, new), přetížení

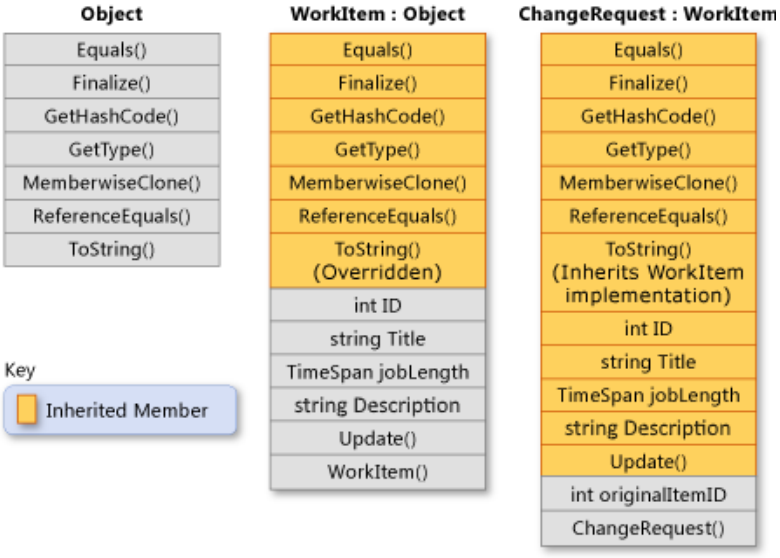
Joseph Soukup – P4

Zapouzdření (Encapsulation)

- Znamená to, že skupina souvisejících vlastností, metod a ostatních členů se považuje za jedinou komponentu nebo objekt.
- Tato vlastnost umožňuje:
    - Svazování dat a funkcionality do objektu, se kterým souvisejí
    - Omezení přístupu k určitým komponentám objektu

Dědičnost (Inheritance)

- Jedna ze tří základních vlastností OOP
- Dědičnost umožňuje vytvořit nové třídy, které znovu použije, rozšíří a mění chování definované v jiných třídách
- Třída jejichž členové jsou zděděni, se nazývají „základní třída“
- třída, která dědí tyto členy, se nazývá „odvozená třída“ -> může mít pouze jednu přímou základní třídu
- Dědičnost je tranzitivní
- Pokud **ClassC** je odvozena z **ClassB, ClassB** je odvozena z **ClassA, ClassC** dědí členy deklarované v **ClassB** a **ClassA**
- Př:
    - Základní třída **Animal,** může mít jednu odvozenou třídu s názvem **Mammal** a jinou odvozenou třídu s názvem **Reptile**
    - **Mammal -> Animal**
    - **Reptile -> Animal**
    - Každá odvozená třída představuje různé specializace



-

C# Kopírovat

```csharp
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }

    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = jobLen;

        // Property originalItemId is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}
```

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

- Podporuje koncept znovu použitelnosti

Polymorfismus

- Často se označuje jako třetí pilíř objektově orientovaného programování, a to po zapouzdření a dědičnosti
    - V době běhu lze objekty odvozené třídy považovat za objekty základní třídy v místech, jako jsou parametry metody a kolekce nebo pole. Pokud tato polymorfismua nastane, deklarovaný typ objektu již není totožný s jeho typem za běhu.
    - Základní třídy mohou definovat a implementovat virtuální metody a odvozené třídy je mohou přepsat, což znamená, že poskytují svou vlastní definici a implementaci
- Umožňují pracovat se skupinami souvisejících objektů jednotným způsobem
- Např.
    - pokud máte aplikaci pro kreslení, která umožňuje uživateli vytvářet různé druhy tvarů na kreslící ploše.
    - V době kompilace neznáte, které konkrétní typy tvarů uživatel vytvoří
    - Aplikace však musí sledovat všechny různé typy tvarů, které byly vytvořeny, a musí je aktualizovat v reakci na akce myši uživatele.
    - Použití polymorfismu:
        - Vytvořte hierarchii třídy, ve které jsou jednotlivé konkrétní třídy tvarů odvozeny ze společné základní třídy
        - Použijte virtuální metodu k vyvolání vhodné metody pro jakoukoli odvozenou třídu prostřednictvím jediného volání metody základní třídy

```
public class Shape
{
    // A few example members
    public int X { get; private set; }
    public int Y { get; private set; }
    public int Height { get; set; }
    public int Width { get; set; }

    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

- Znamená možnost záměnného použití více tříd, které mají stejná data a metody ale implementuje je jinými způsoby
- Je to schopnost objektu nebo reference se chovat jinak v různých instancích
- Podporuje koncepty:
    - Přepsání a překrytí funkce (overriding)
    - Přetížení funkce (overloading)

```
static int PlusMethod(int x, int y)
{
  return x + y;
}

static double PlusMethod(double x, double y)
{
  return x + y;
}

static void Main(string[] args)
{
  int myNum1 = PlusMethod(8, 5);
  double myNum2 = PlusMethod(4.3, 6.26);
  Console.WriteLine("Int: " + myNum1);
  Console.WriteLine("Double: " + myNum2);
}
```

Polymorfismus časná a pozdní vazba

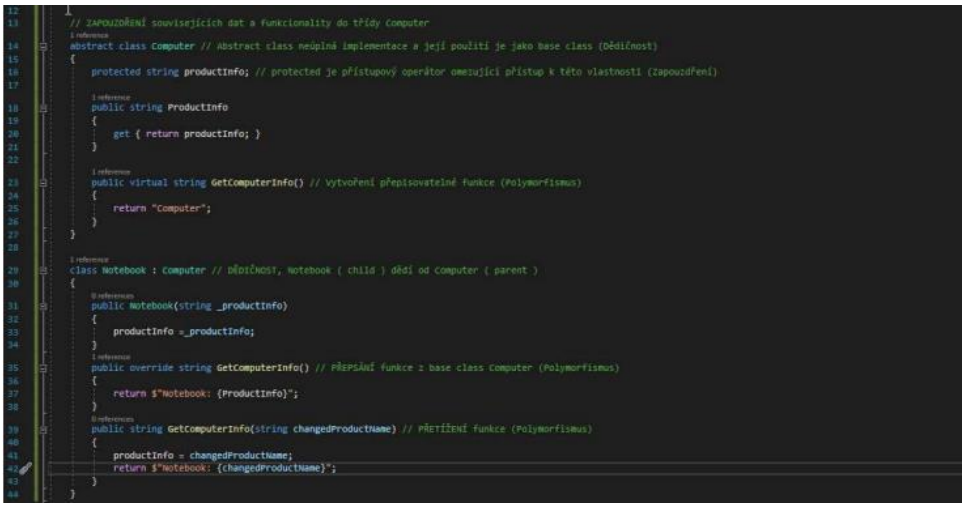
- **Časná vazba:** (statická vazba)
  - Kompilátor v době překladu přesně ví, jaký objekt, vlastnosti a metody se budou volat
  - Rychlejší a menší hrozba chyb za běhu programu
- **Pozdní vazba:** (dynamická vazba)
  - Typ objektu s jeho metodami a vlastnostmi se zjistí až za běhu programu
  - Používá se u dědičnosti
  - Použití pozdní vazby může signalizovat použitím klíčového slova **virtual**

Přepsání a překrytí (overriding)

- Implementace polymorfismu v dědičnosti, umožňuje nám změnit chování metody, která byla zděděná ze super class v sub class
- V C# použijeme:
  - **Virtual** při deklaraci metody v super class. Tím umožníme všem sub class upravit chování metody pro své potřeby
  - **Override** při deklaraci metody v sub class, což nám umožní přepsat virtual metodu ze super class

Přetížení (overloading)

- **Přetížení metody:**
  - Běžná implementace polymorfismu ve stejné scope.
  - Umožňuje nám změnit implementaci metod se stejným názvem ale jinou signaturou.
  - Jiná signatura vznikne změnou parametrů, což může být změna počtu parametrů, jejich název nebo pořadí.
- **Přetížení operátoru:**
  - Koncept přetížení metody se dá použít i na operátory
  - To nám umožňuje rozšířit nebo změnit funkcionalitu předdefinovaných operátorů


// ZAPOUZDŘENÍ souvisejících dat a funkcionality do třídy Computer
abstract class Computer // Abstract class neúplná implementace a její použití je jako base class (Dědičnost)
protected string productInfo; // protected je přístupový operátor omezující přístup k této vlastnosti (zapouzdření)
public string ProductInfo
get { return productInfo; }
public virtual string GetComputerInfo() // Vytvoření přepisovatelné funkce (Polymorfismus)
return "Computer";
class Notebook : Computer // DĚDIČNOST, Notebook (child) dědí od Computer (parent)
public Notebook(string _productInfo)
productInfo =_productInfo;
public override string GetComputerInfo() // PŘEPSÁNÍ funkce z base class Computer (Polymorfismus)
return $"Notebook: {ProductInfo}";
public string GetComputerInfo(string changedProductName) // PŘETÍŽENÍ funkce (Polymorfismus)
productInfo = changedProductName;
return $"Notebook: {changedProductName}";