impera

Objektorienterad programmering med C#

Kursträff 6

Arv, interfaces, generics, design patterns

Dagens föreläsning

1. **Klasser i C#**
    - Egenskaper
    - Klassmetoder och -egenskaper
    - Overloads
    - Arv
        - Överlagring
        - Abstrakta klasser
    - Interfaces (gränssnitt)
    - Polymorfism
    - Generics
2. **Design patterns**
    - SOLID
    - Clean code

Egenskaper (*properties*)

```
public class Person {
    private string _name;
    public string Name {
        get { return _name; }
        set { _name = value; }
    }
}

// Auto-property:
public class Person {
    public string Name { get; set; }
}
```

- Properties är ett sätt att kapsla in fält på en klass.
  - Accessor-metoder - get och set
    - Special variabeln value innehåller värdet som skickas in:
      - `person.Name = "Sven"; // value == "Sven"`
  - Gör det möjligt att
    - Validera innan värden sätts
    - Skapa egenskaper som räknar ut något, t.ex.

Egenskaper (*properties*)

```
public class Company {
    public List<Person> Employees { get; set; }
    public int NumberOfEmployees {
        get { return Employees.Count(); }
        // Ingen set accessor!
    }
}

// …

var company = new Company();
var numEmployees = company.NumberOfEmployees;
```

Klassmetoder och -egenskaper

```
public class Person {
    internal static string UnknownName = "N/A";
    public string Name { get; set; }
    public Role Role { get; set; }

    public Person(string name = null) {
        // Klassegenskapen måste hämtas från klassen, även
        // i klassen själv:
        this.Name = name ?? Person.UnknownName;
    }

    public static string JoinNames(string[] names) {
        return names.join(" ");
    }
}
```

- Statiska metoder kallas också klassmetoder
  - Egenskaper kan också vara statiska
- Statiska metoder finns på klassen, *inte* på ett enskilt objekt
  - ```
    var p = new Person();
    p.Name = Person.JoinNames(new [] { "Kalle", "Markus", "Räisänen"} );
    ```
    ```

“Overloads"

```
public class Person {
    public string FormatName() {
        return this.Name;
    }

    public string FormatName(string prefix) {
        return prefix + this.FormatName();
    }

    public string FormatName(string prefix, string suffix) {
        return this.FormatName(prefix) + suffix;
    }
}
```

- Man kan ha flera varianter av samma metod
  - De nya varianterna kallas “overloads”
- Varianterna måste ta olika (typ eller antal) parametrar
- Varianter av metoderna kan anropa varandra

Arv

```
public class Person {
    internal static UnknownName = "N/A";
    public string Name { get; internal set; }
    public Role Role { get; protected set; }

    public Person(string name = null) {
        this.Name = name ?? Person.UnknownName;
    }
}
```

```
public class Student : Person  {
    public Student(string name = null)
        : base(name) {
        this.Role = Role.Student;
    }
}
```

- Arv skrivs med ":":
  - `class SubKlass : SuperKlass`
- Vi kallar den ärvande klassen för "subklass" och klassen den ärver från "superklass"
- En klass kan bara ärva från *en* superklass
- För att anropa superklassens konstruktor använder vi base vid underklassens konstruktor:
  - `public KlassNamn() : base() { ...`

- **Synlighetsmodifierare:**
  - **`public`** - Synlig överallt
  - **`private`** - Synlig bara i denna klass
  - **`protected`** - Synlig i denna och ärvande klasser
  - **`internal`** - Synlig i denna assembly
- Synlighetsmodifierare kan också användas på accessor-metoder

Arv: överlagring

```
public class Person {
    // …
    public virtual string FormatName() {
        return this.Name;
    }

    public string TruncateName() {
        return this.Name.Substr(0, 10);
    }
}
```

- Metoder och egenskaper som skrivs med `virtual` går att överlagra (override)
- Överlagring innebär att en subklass ersätter en metod i superklassen med en egen variant
  - `override`

```
public class Student : Person  {
    // …
    public override string FormatName() {
        return base.FormatName() + " (Student)";
    }

    public new string TruncateName() {
        // ...
    }
}
```

- I subklassen kan man komma åt superklassens variant av metoden via nyckelordet `base`.
- Metoder som inte är virtual kan ersättas i subklassen med nyckelordet new
  - Det kallas *method hiding*
  - Ställer till problem med polymorfism och bryter mot SOLID

Arv: abstrakta klasser

```
abstract public class Shape {
    public int Side { get; set; }

    abstract public int Area();
}
```

```
public class Square : Shape  {
    public override int Area() {
        return Side * Side;
    }
}
```

- Klasser kan vara *abstrakta*
  - Metoder i abstrakta klasser ges bara “signaturer” i klassen
    - namn, parametrar, returtyp
  - De definieras i ärvande klass
    - Subklassen har metodkropp
  - En abstrakt klass kan inte instansieras
    - Det går inte skapa ett objekt av typen Shape i exemplet

- Klasser som ärver abstrakta klasser måste implementera alla abstrakta metoder i superklassen
  - Liksom med överlagring används override
- Abstrakta klasser kan ha metoder och egenskaper som inte är abstrakta
  - Dessa behöver inte implementeras i subklasser

Interfaces (gränssnitt)

```
interface IPerson {
    string FormatName();
}
```

```
public class Person : IPerson  {
    public string FormatName() {
        return this.Name;
    }
}
```

- Ett gränssnitt är en slags mall för en klass
  - Skapas med nyckelordet “interface”
- Gränssnitt deklarerar “signaturer”:
  - Vilka metoder som ska finnas
  - Vilka parametrar dessa ska ta
  - Vilka returtyper de ska ha

- Klasser *implementerar* gränssnitt
  - Implementationen definierar metodkropp för de signaturer som gränssnittet har deklarerat
- En klass kan implementera flera gränssnitt samtidigt:
  - `class Person : IPerson, IEmployee`

Polymorfism

- “Flerformighet”
- En instans av en subklass kan ehandlas som om den vore en instans av en superklass
  - Även klasser som implementerar ett gränssnitt kan behandlas som “instanser” av gränssnittet
- Om vi har en klass Student som ärver från Person, så kan vi lagra en instans av Student i en variabel av typen Person:
  - `Person p = new Student(); // Student : Person`
- Det är extra användbart i samlingar:
  - Vi kan ha en samling av superklassen, och lagra instanser av subklasser i den

Polymorfism

```
public class Person {
    public string Name { get; set; }
}

public class Student : Person {
    public string StudentKod { get; set; }
}

public class Teacher : Person {
    public int EmployeeId { get; set; }
}
```

```
var personList = new List<Person>();

personList.AddRange(new [] {
    new Student(),
    new Teacher(),
    new Person()
});

foreach (var p in personList) {
    // p är en instans av Person-klassen
    Console.WriteLine(p.Name);
}
```

Polymorfism

```
interface IHtmlTag {
    string ToHtml();
}

public class P : IHtmlTag {
    public string Content { get; set; }

    public P(string content) {
        this.Content = content;
    }

    public string ToHtml() {
        return "<p>" + Content + "</p>";
    }
}

public class Br : IHtmlTag {
    public string ToHtml() {
        return "<br/>";
    }
}
```

```
var tags = new List<IHtmlTag> {
    new P("Text"),
    new Br(),
    new Br()
    // …
};

foreach (var t in tags) {
    // Vi vet att alla klasser som
    // implementerar IHtmlTag
    // måste ha en ToHtml()-metod
    Console.WriteLine(t.ToHtml());
}
```

Generics

- Generics innebär att vi kan skapa klasser med “platshållare” för typer för dess metoder, egenskaper, osv
  - Man bestämmer typ vid instansiering
- Vi har redan stött på generics:
  - `List<T>`
    - T är en platshållare
  - `Dictionary<TKey, TValue>`
    - `TKey` och `TValue` är båda platshållare

Generics

```
public MyList<T> {
    private int lastIndex = 0;
    private T[] items;

    public MyList() {
        items = new T[1024];
    }

    public void Add(T obj) {
        if (lastIndex < 1024) {
            items[lastIndex++] = obj;
        }
    }

    public T Get(int index) {
        return index < lastIndex
            ? items[index]
            : null;
    }
}
```

```
var list = new MyList<string>();
```

- Blir som att T ersätts med `string` i klassen

impera

Generics

- Man kan ärva en generic klass och på det sättet skapa en icke-generisk version av den
  - Om man till exempel vill skapa en List bara för string kan man skriva:
    - `public class StringList : List<string> { }`
  - Då kan man sedan använda den precis som en `List<string>`.
    - `var list = new StringList { "Lorem", "Ipsum" };`
  - Man kan naturligtvis också lägga till egna metoder och egenskaper till sin klass:
    -
```
public class DecimalList : List<decimal> {
    public int NumPositive {
        get {
            int count = 0;
            foreach (var num in this) {
                if (num > 0) { count++; }
            }
            return count;
        }
    }
}
```

Generics: constraints

- När man skapar en generic klass kan man begränsa vilka typer den kan användas med
  - Begränsa till vissa klasser:
    - `public class MyList<T> where T : Animal {}`
      T får bara vara Animal eller subklasser till Animal
    - `public class MyList<T> where T : IComparable {}`
      T måste vara en klass som implementerar gränssnittet IComparable
    - `public class MyList<T> where T : IComparable, IEnumerable {}`
      T måste vara en klass som implementerar gränssnitten IComparable och *IEnumerable*
  - Begränsa till referenstyper (klasser, interface, delegat, arrayer):
    - `public class MyList<T> where T : class {}`
  - Begränsa till typer med parameterlöskonstruktor:
    - `public class MyList<T> where T : new() {}`
      T får bara vara typer som man kan skapa med new TypNamn()

Generics: constraints

- Man kan också använda constraints för att skapa en begränsad version av en annan generisk klass
  - `public class MyList<T> : List<T> where T : IComparable {}`
    MyList<T> blir som en List<T> där man bara kan skapa listor av klasser som implementerar IComparable

Generics: metoder

- Man kan också göra en metod generisk:
  - ```
    public class StringList : List<string> {
        public T GetCount<T>() {
            return (T)this.Count; // casta resultatet av Count till T
        }
    }

    public static class Helper {
        public static string Stringify<T>(T obj) where T : object {
            // Alla klasser som ärver från object har ToString
            return obj.ToString();
        }
    }
    ```
  - ```
    var list = new StringList { "One", "Two", "Three" };
    var count = list.GetCount<int>(); // count är int
    var str = Helper.Stringify(8); // <int> är implicit!
    ```

Design patterns

- Design patterns är mönster för att lösa återkommande problem
  - Hjälper till att göra kod mer lättförståelig
  - Gör koden lättare att förändra

SOLID

- S: SRP (Single responsibility principle)
  - Varje klass ska vara ansvarig för en sak
- O: OCP (Open closed principle)
  - En klass ska vara öppen för utökning men stängd för förändring
    - Om vi behöver stödja någon ny variant ska det vara en ny klass som utökar den existerande
- L: LSP (Liskov substitution principle)
  - Instanser av klasser som ärver en superklass ska kunna behandlas som en instans av superklassen
- I: ISP (Interface segregation principle)
  - Tvinga inte klasser att implementera metoder den inte behöver
    - Lägg inte till metoder i gränssnitt som inte alla klasser som implementerar gränssnittet behöver
- D: DIP (Dependency inversion principle)
  - Använd så långt som möjligt gränssnitt istället för specifika klasser

Single Responsibility Principle

- Varje klass ska vara ansvarig för en sak
- Säg att vi har en användarklass:

```
public class User {
    public string Username { get; set; }
    public string Password { get; set; }
    public bool LogIn(string username, string password) {
        if (username == Username && password == Password) {
            return true;
        } else {
            Console.WriteLine("Log in failed!");
            return false;
        }
    }
}
```

- Diskutera: Vad är problemet?

Single Responsibility Principle

```
public class User {
    public string Username { get; set; }
    public string Password { get; set; }
    public bool LogIn(string username, string password) {
        if (username == Username && password == Password) {
            return true;
        } else {
            Console.WriteLine("Log in failed!");
            return false;
        }
    }
}
```

- En användarklass har inte något med att göra med att skriva ut felmeddelanden
- Skapa en ny klass som hanterar felmeddelanden istället och använd den antingen i User eller i koden som använder User.LogIn

Open Closed Principle

- En klass ska vara öppen för utökning men stängd för förändring
- Om vi har en klass som heter Person:

```
public class Person {
    public string Name { get; set; }
}
```

- Om vi då vissa personer är anställda och ska ha ett anställningsnummer så lägger vi inte till det på `Person` utan ärver från den:

```
public class Employee : Person {
    public int EmployeeNumber { get; set; }
}
```

Liskov Substitution Principle

- Instanser av klasser som ärver en superklass ska kunna behandlas som en instans av superklassen
- Om vi har en klass som heter File som kan öppnas och sparas:

```
public class File {
    public virtual void LoadFile() { }
    public virtual void SaveFile() { }
}
```

- Då vill vi inte skapa en ny klass som ärver från File som *inte* kan sparas:

```
public class ReadOnlyFile : File {
    public override void SaveFile() {
        Console.WriteLine("Can’t save ReadOnlyFiles!");
    }
}
```

Dependency Inversion Principle

- Tvinga inte klasser att implementera metoder den inte behöver
  - Som SRP för gränssnitt: lägg inte till metoder som inte har med gränssnittet att göra

```
interface IPerson {
    string Name { get; set; }
    string Username { get; set; }
    bool ValidateUsername();
}

class User : IPerson {
    string Name { get; set; }
    string Username { get; set; }
    bool ValidateUserName() {
        return Username.Length > 6;
    }
}
```

```
class Employee : IPerson {
    string Name { get; set; }
    string Username { get; set; }
    bool ValidateUserName() {
        return true;
    }
}
```

- Både `Username` och `ValidateUsername()` bryter mot ISP

Interface Segregation Principle

- Använd så långt som möjligt gränssnitt istället för specifika klasser
  - (Förenklat uttryckt)

```
interface IPerson {
    string Name { get; set; }
    string FormatName();
}
class User : IPerson {
    string Name { get; set; }
    string Username { get; set; }

    string FormatName() {
        return Name + " - " + Username;
    }
}
class Employee : IPerson {
    string Name { get; set; }
    string Id { get; set; }

    string FormatName() {
        return Name + "(" + Id + ")";
    }
}
```

```
class PersonPrinter {
    void PrintUser(User u) {
        Console.WriteLine(u.FormatName());
    }
    void PrintEmployee(Employee e) {
        Console.Writeline(e.FormatName());
    }
}

class PersonPrinter {
    void PrintPerson(Person p) {
        Console.WriteLine(p.FormatName());
    }
}
```

SOLID

- S: SRP (Single responsibility principle)
  - Varje klass ska vara ansvarig för en sak
- O: OCP (Open closed principle)
  - En klass ska vara öppen för utökning men stängd för förändring
- L: LSP (Liskov substitution principle)
  - Instanser av klasser som ärver en superklass ska kunna behandlas som en instans av superklassen
- I: ISP (Interface segregation principle)
  - Tvinga inte klasser att implementera metoder den inte behöver
- D: DIP (Dependency inversion principle)
  - Använd så långt som möjligt gränssnitt istället för specifika klasser

Man kan säga att alla SOLID-principer är varianter av SRP för olika kontexter

Clean code

- En samling principer som går ut på att kod ska vara lätt att läsa
- Bland annat handlar det om:
  - Meningsfulla namn
    - Säg vad en variabel innehåller:
      - `var d = 32; // elapsed time in days`
      - `var elapsedDays = 32;`
    - Klassnamn bör vara substantiv, t.ex. Day, Car, TimeParser. Undvik utfyllnadsord som Data, Info (alltså User inte UserInfo eller UserData).
    - Metodnamn bör vara verb eller verbfraser, t.ex. GetTimeOfDay(), UpdateAccount().
    - Använd konsekvent terminologi. Om en metod för att hämta data heter Fetch() bör alla metoder som hämtar data heta något med Fetch.
  - Små metoder som gör en sak
    - Dela upp stora metoder i många små
    - Följ SOLID
  - Undvik onödiga kommentarer

Clean code: Factory-method pattern

- Konstruktorer kan overload-as:

```
public class Time {
    public int Hours;
    public int Minutes;
    public int Seconds;

    public Time(int h) {
        this.Hours = h;
    }
    public Time(int h, int m) {
        this.Hours = h;
        this.Minutes = m;
    }
    // osv ...
}
```

- Det kan lätt bli lite svårläst:

```
var t1 = new Time(1);
var t2 = new Time(1, 30);
// osv…
```

- Det kan göras tydligare med statiska “factory methods”:

```
public class Time {
    // Samma kod som tidigare

    public static Time FromHours(int h) {
        return new Time(0);
    }
    public static Time FromMinutes(int m) {
        return new Time(0, m);
    }
    public static Time FromHoursAndMinutes(int h, int m) {
        return new Time(h, m);
    }
}
```

- Vi kan sen skapa nya instanser genom att använda factory metoder:

```
var t1 = Time.FromHours(0);
var t2 = Time.FromHoursAndMinutes(1, 30);
```

impera