Module 04

”Object-oriented Programming in C#”

TEKNOLOGISK INSTITUT

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`

Object-Oriented Modeling

- Attempts to realistically reflect (part of) the real-world
- Introduced as a mechanism to ease modeling of simulation problems
- Slowly but steadily adopted into programming languages since 1973

- Abstraction is a crucial technique in this endeavor
  - Focus on important aspects
  - Disregard irrelevant aspects
  - "Selective ignorance"
  - Makes complex things simple!

- Main concepts include *Classes* and *Objects*

The Concept of Classes

- A class in effect classifies <u>abstract</u> or <u>concrete</u> things!

- Philosophers
  - Use artifacts of human classification
  - Classify concepts based upon common characteristics, behavior, and attributes
  - Create descriptions and names of such classifications

- Object-oriented programmers
  - Classify concepts using specific syntactic constructs describing behavior and attributes
  - Define data structures including both data and methods

The Concept of Objects

- Classes are “blueprints” for objects
  - An object is an instance of a class

- Objects have
  - Identity
    - Unique, Distinguishable
  - State
    - Setting, Data
  - Behavior
    - Performing operations modifying the state

- In (sloppy) everyday language the same vocabulary is often used for both the object and the class from which it originates

Examples of Classes and Objects

Structs Vs. Classes

- Structs are “blueprints” for values
  - No distinguishable identity
  - No inaccessible state
  - No “behavior”

- Classes are “blueprints” for objects
  - Distinguishable identity
  - State can be inaccessible
  - Behavior central to object

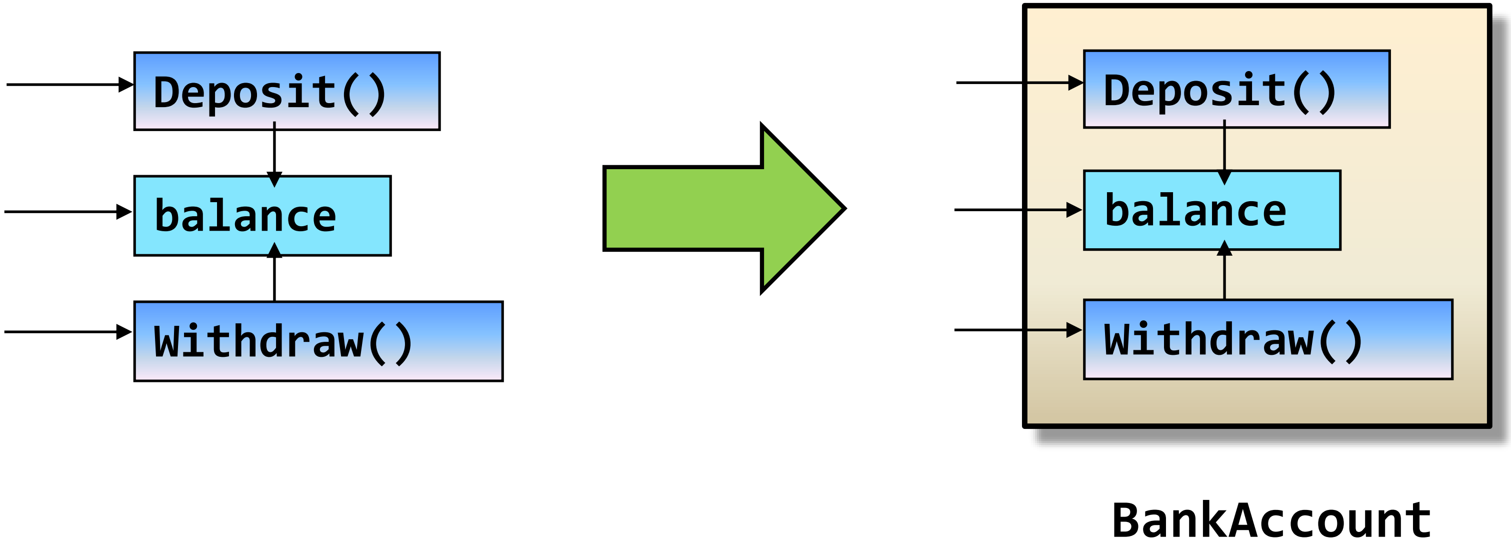
Agenda

TEKNOLOGISK
INSTITUT

- Introducing Object-Oriented Programming
- **First Pillar of OOP: Encapsulation**
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- **`System.Object`**

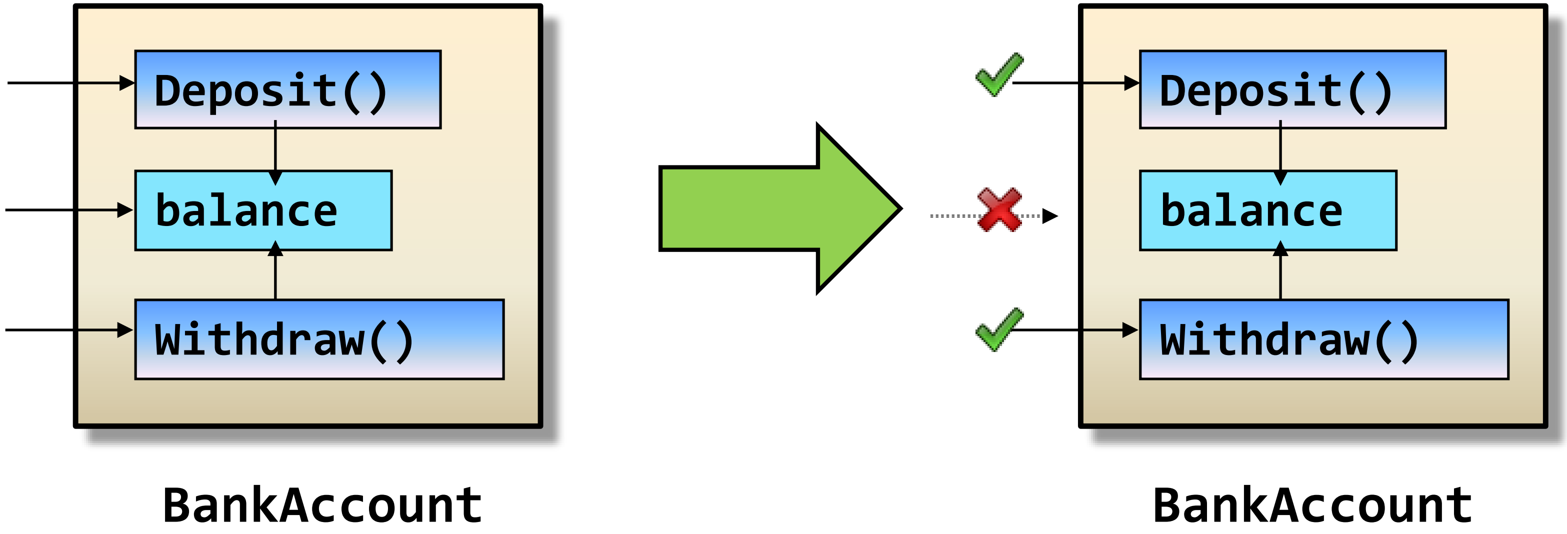
Introducing Encapsulation

- Grouping related ideas in a single unit



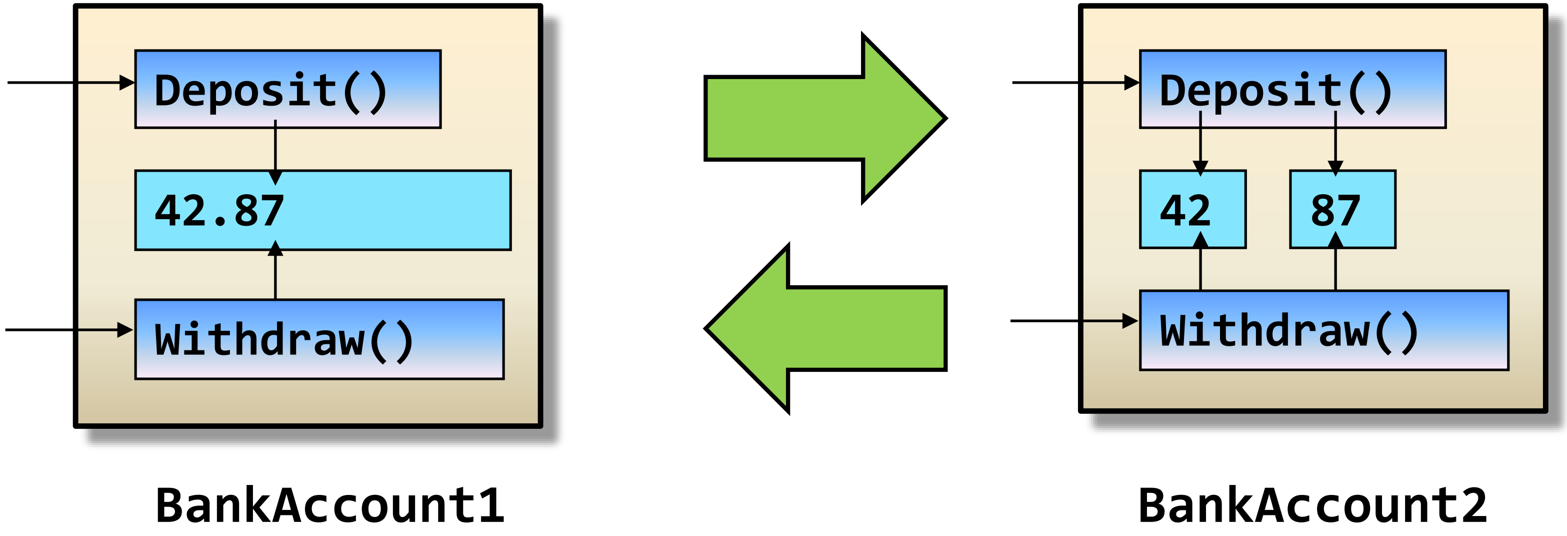
Introducing Encapsulation (2)

- The packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only through the objects' interface



Introducing Encapsulation (3)

- The ability to hide internal detail to the outside
- Ability to reuse objects without internal representation



The Three Pillars of OOP

- Encapsulation
  - The grouping of related ideas in a single unit
  - The packaging of operations and attributes representing state into an object type so that state is accessible or modifiable only through the objects' interface
  - The ability to hide internal detail to the outside
  - Ability to reuse objects without internal representation
- Inheritance
- Polymorphism

Agenda

TEKNOLOGISK
INSTITUT

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- **Creating Classes and Objects**
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`

Defining Classes

- Classes are defined using the `class` keyword

```
class Car
{
   public string petName;
   public int currentSpeed;

   public void PrintState()
   {
      Console.WriteLine( "{0} is going {1} km/h",
         petName,
         currentSpeed );
   }
   public void SpeedUp( int delta )
   {
      currentSpeed += delta;
   }
}
```

Allocating Objects

- Objects are instantiated by the new keyword

```
Car myCar = new Car();
myCar.petName = "Goofy";

for( int i = 0; i < 5; i++ )
{
   myCar.SpeedUp( 10 );
   myCar.PrintState();
}
```

- Objects are not allocated in memory until they are "new'ed"

```
Car myCar;
myCar.petName = "Goofy";
```

Default Constructor

- Every class has a *default constructor* method supplied out-of-the-box
  - Takes no arguments and has no return type
  - Sets all field data to a default value
- The constructor is invoked when an object is allocated with **`new`**
- The default constructor can be redefined

```
class Car
{
   public string petName;
   public int currentSpeed;

   public Car()
   {
      petName = "Chuck";
      currentSpeed = 10;
   }
}
```

Custom Constructors

- Any set of overloaded custom constructors can be defined

```
class Car
{  ...
   public Car( string pt )
   {
      petName = pt;
   }
   public Car(string pn, int cs)
   {
      petName = pn;
      currentSpeed = cs;
   }
}
```

```
Car chuck = new Car( "Chuck" );
Car goofy = new Car( "Goofy", 87 );

chuck.PrintState();
goofy.PrintState();
```

- Note: When you define a custom constructor, the compiler silently removes the built-in default constructor!

The this Keyword

- In any class the **this** keyword is a reference to the current object
- It can be used to e.g. resolve naming conflicts

```
class Car
{
   public string petName;

   public Car( string petName )
   {
      this.petName = petName;
   }
}
```

- Local variables overshadow member variables
- Useful with IntelliSense

Chaining Constructors

- Constructors can be chained using **`this`**
- In this way the core construction code can be kept non-duplicated
  - Often there is a central initialization method of sorts

```
public Car() : this( "Chuck" )
{
}
public Car( string petName ) : this( petName, 0 )
{
}
public Car( string petName, int currentSpeed )
{
   // This is the central initialization code
   this.petName = petName;
   this.currentSpeed = currentSpeed;
}
```

Revisiting Optional Arguments

- The optional arguments of Module 03 can also be applied for constructors

```
public Car( string petName = "Chuck", int
currentSpeed = 0 )
{
   // This is the central initialization code
   this.petName = petName;
   this.currentSpeed = currentSpeed;
}
```

```
Car alice = new Car( "Alice", 30 );
Car bob = new Car( "Bob" );
Car chuck = new Car( currentSpeed: 50 );
```

- Carefully chosen default values usually reduce the number of necessary constructors

Partial Classes

- The implementation of a class can be divided into multiple .cs-files

```
// Car.Constructors.cs
partial class Car
{
   public Car( string pt )
   {
      petName = pt;
   }
   public Car(string pn, int cs)
   {
      petName = pn;
      currentSpeed = cs;
   }
}
```

```
// Car.cs
partial class Car
{
   public string petName;
   public int currentSpeed;

   public void SpeedUp( int delta )
   {
      currentSpeed += delta;
   }
}
```

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`

Access Modifiers

Access Modifier	Meaning…
public	No access restrictions
private	Can only be accessed by the defining type
protected	Can only be accessed by the defining type and its derived types
internal	Accessible only within the current assembly defining the type
protected internal	Protected + Internal; Accessible only within the current assembly defining the type as well as in derived types

Default Access Modifiers

- Members are implicitly private
- Types are implicitly internal

```
namespace Devices
{
   class Radio    // internal class
   {
      Radio()     // private constructor
      {
      }
   }
}
```

- Good style to declare access modifier explicitly (even if default)

Access Modifiers and Nested Types

- Nested types can be access-modified as well

```
public class Tv
{
   private enum Encoding { Mpeg2, Mpeg4 }; // Only
visible inside Tv

   public Tv()
   {
   }
}
```

- Top-level types cannot be private!

A Matter of Style and Taste

- There are no mandatory rules for the nomenclature of classes, members etc.
- Best approach is to follow Microsoft ☺
  - Classes and other Types are PascalCase
  - Methods and Properties are PascalCase
  - Public member variables are PascalCase
  - Parameters are camelCase
- Religious issues
  - Private member variables are _camelCase
  - Member variables at top of class definition
    - Except… ☺
  - …

```
class Car
{
   public string PetName;
   private int _currentSpeed;

   public void SpeedUp(int delta)
   {
      ...
   }
}
```

Quiz: Classes – Right or Wrong?

```
class Car
{
   public string PetName;
   public int CurrentSpeed;
}
```

✓

```
Car c;
c.PetName = "Beardyman";
```

✗

```
Car c = new Car();
c.PetName = "Beardyman";
```

✓

```
class Person
{
   string Name;

   public void Person(string name)
   {
      this.Name = name;
   }
}
```

✗

```
Person p = new Person("Dude");
```

✓

```
Person p = new Person();
```

✗

```
Person p = new Person("Dude");
p.Name = "Homie";
```

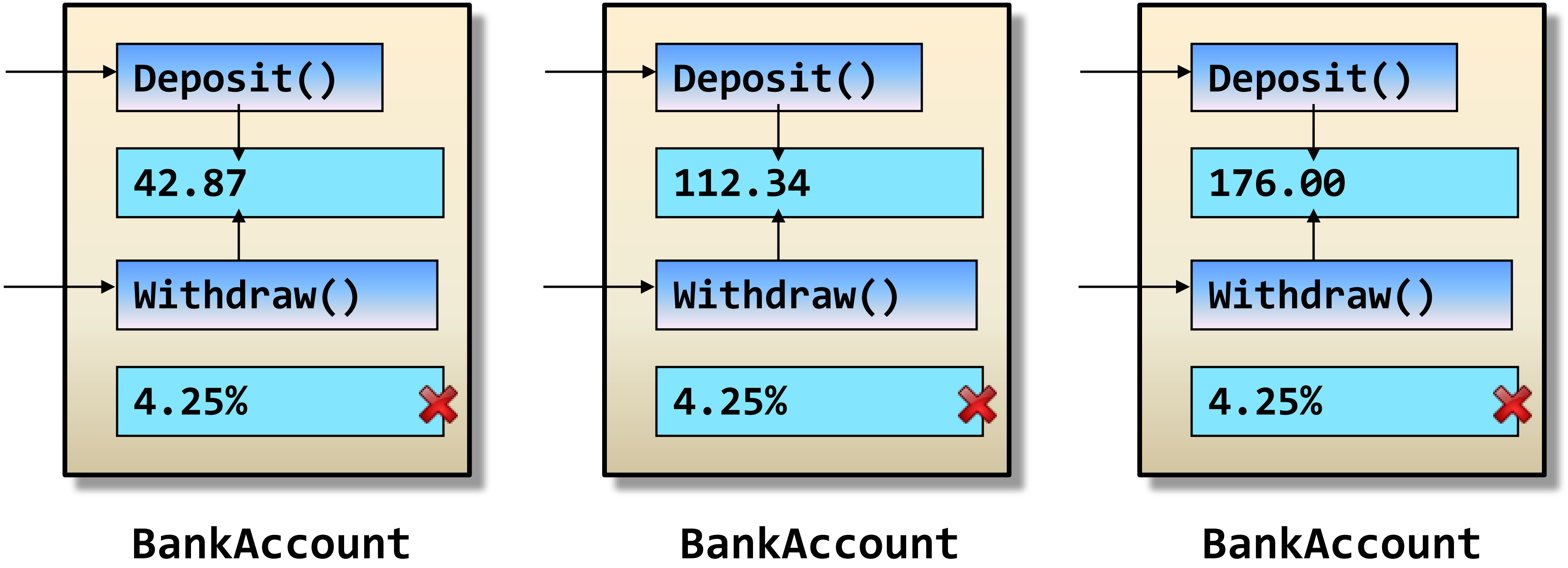
✗

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- **Static Classes and Members**
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`

Introducing Static Data

- Static data captures information shared between all the objects of a class



Static Data

- With instance data each object maintains an independent copy
- Class data can be *static*, i.e. shared among all instances

```
class BankAccount
{
   private decimal _currentBalance;
   public static decimal CurrentInterestRate = 0.04m;

   public BankAccount( decimal balance )
   {
      _currentBalance = balance;
   }
}
```

- Refers to the same physical in-memory location!

Static Methods

- Static data should be manipulated by static methods

```
class BankAccount
{
   ...
   public static decimal CurrentInterestRate = 0.04m;

   public static void SetInterestRate( decimal interestRate )
   {
      CurrentInterestRate = interestRate;
   }
}
```

- Invoke static methods via class name instead of instance name!

```
BankAccount.SetInterestRate( 0.06m );
```

Static Constructors

- Initializing static data should be done in static constructors

```
class BankAccount
{
   public static decimal CurrentInterestRate;

   static BankAccount()
   {
      CurrentInterestRate = 0.04m; // This could be dynamic!
   }
}
```

- Only one static constructor for each class
- Has no access modifier and no parameters
- Invoked by the runtime system before first instance constructor
- Invoked exactly once regardless of number of objects created

Static Classes

TEKNOLOGISK INSTITUT

- Classes themselves can also be static

```
static class TimeUtility
{
   public static void PrintTime()
   {
      Console.WriteLine( DateTime.Now.ToShortTimeString() );
   }
   public static void PrintDate()
   {
      Console.WriteLine( DateTime.Today.ToShortDateString() );
   }
}
```

- Cannot be instantiated

```
TimeUtility tu = new TimeUtility();
```

- Can only contain static fields and methods

Agenda

TEKNOLOGISK
INSTITUT

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- **Properties and Initializers**
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`

Properties

- Encapsulation is achieved by *Properties*

```
class Button
{
   public string Caption
   {
      get { return _caption; }
      set { _caption = value; }
   }
   private string _caption;
}
```

```
Button button = new Button();
button.Caption = "Click!!";

Console.WriteLine( button.Caption );
```

- Two specific accessors
  - get is invoked when retrieving the value
  - set is invoked when setting the value

Visibility of Get/Set

- Access modifiers can be set for **get** and **set** separately

```
class Button
{
   public string Caption
   {
      get { return _caption; }
      private set { _caption = value; }
   }
   private string _caption;
}
```

```
Button button = new Button();
Console.WriteLine( button.Caption ); ✓

button.Caption = "Click!!";          ✗
```

Read-Only and Write-Only Properties

- Property can be made read-only by omitting **set**
- Property can be made write-only by omitting **get**

```
class Button
{
   public string Caption
   {
      get { return _caption; }
      // No set!
   }
   private string _caption;
}
```

```
class Login
{
   public string Password
   {
      // No get!
      set { _password = value; }
   }
   private string _password;
}
```

- Properties can also be static

Defining Automatic Properties

- Automatic properties ease the burden of defining “trivial” properties

```
class Car
{
   public string PetName
   {
      get { return _petName; }
      set { _petName = value; }
   }
   private string _petName = string.Empty;
}
```

```
class Car
{
   public string PetName { get; set; }
}
```

Default Values of Automatic Properties

TEKNOLOGISK INSTITUT

- Default value of an automatic property is the usual "zero-whitewash"
  - Reference types are null
  - Integers are 0
  - Booleans are false
  - …
- If any other default value is required, it must be set in the constructor

```
class Car
{
   public Car() { PetName = "Goofy"; }
   public string PetName { get; set; }
}
```

Restricting Access to Automatic Properties

- Automatic properties cannot be read-only or write-only by omitting get or set!

```
class Car
{                                              ✗
   public string PetName { get; }
}
```

- Use access modifiers on get or set

```
class Car
{
   public string PetName { get; private set; } ✓
}
```

- Note: Neither get nor set can be more visible than the parent property

Object Initializer Syntax

- Object initializer syntax can be used to assign values for public properties and fields during construction

```
Point p = new Point { X = 42, Y = 87 };
Console.WriteLine( "p is {0}", p );
```

- Custom constructors can be invoked as well

```
Point q = new Point( 16, 24 ) { X = 112 };
Console.WriteLine( "q is {0}", q );
```

- Object initializers execute after constructors
- Object initializers can initialize any subset of available properties and fields

Initializing Inner Types and Collections

- Inner types can now be conveniently initialized

```
public class Rectangle
{
   public Point TopLeft { get; set; }
   public Point BottomRight { get; set; }
   ...
}
```

```
Rectangle r = new Rectangle
{
   TopLeft = new Point { X = 10, Y = 10 },
   BottomRight = new Point { X = 90, Y = 90 }
};
Console.WriteLine( r );
```

Methods vs. Properties

- Properties are somewhere in between public member variables and methods
- Methods
  - Defined and invoked using parenthesis
  - Might take parameters
- Properties
  - Defined and invoked without parenthesis
  - No additional parameters: Gets or sets a single value

```
class BankAccount
{   ...
    public decimal GetBalance()
    {
        return _balance;
    }
}
```

```
BankAccount ba = ...;
decimal d = ba.GetBalance();
```

```
class BankAccount
{   ...
    public decimal Balance
    {
        get { return _balance; }
    }
}
```

```
BankAccount ba = ...;
decimal d = ba.Balance;
```

Constant Data

- Data is deemed constant by using the **`const`** keyword

```
class MyMathClass
{
   public const double Pi = 3.14;
}
```

```
Console.WriteLine( MyMathClass.Pi ); ✓

MyMathClass.Pi = 22 / 7;             ✗
```

- Such data cannot be changed!

- Curious fact: Constant fields are implicitly static

Read-only Data

- Read-only data can <u>only be set in constructors</u>

```
class MyMathClass
{
   public MyMathClass()
   {
      if( DateTime.Today.Day % 2 == 0 )
      {
         TodaysPi = 3.14;
      }
      else
      {
         TodaysPi = 22.0 / 7;
      }
   }
   public readonly double TodaysPi;
}
```

```
MyMathClass m = new MyMathClass();
Console.WriteLine( m.TodaysPi );  ✓

m.TodaysPi = 4.00;                ✗
```

```
public void SetTodaysPi( double tp )
{
   TodaysPi = tp;
}
```

✗

Quiz: Properties and Static Members – Right or Wrong?

✓
```
class Car
{
   public static int SpeedLimit;
   public string PetName;
   public int CurrentSpeed;
}
```

✓
```
class Point
{
   public int X { get; set; }
   public int Y { get; set; }
}
```

✗
```
class Person
{
   public int Id { private get; }
}
```

✗
```
Car c;
c.SpeedLimit = 50;
```

✗
```
Car c = new Car();
c.SpeedLimit = 50;
```

✓
```
Car.SpeedLimit = 50;
```

✓
```
Point p = new Point();
p.X = 42;
p.Y = 87;
```

✓
```
Point p = new Point{ X = 42 };
```

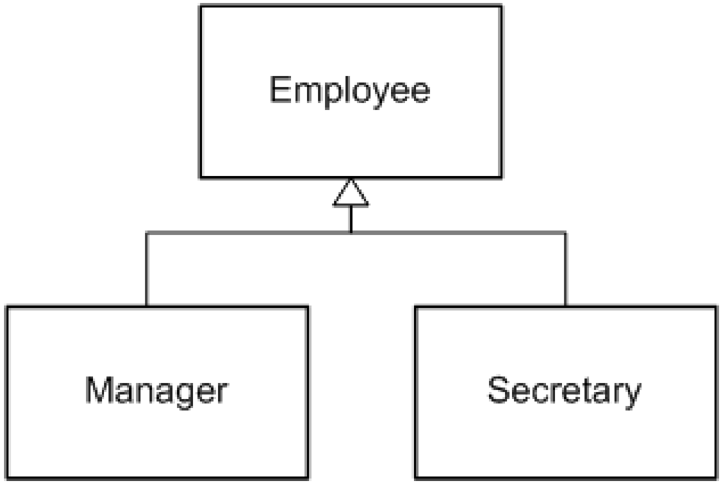
Agenda

TEKNOLOGISK
INSTITUT

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- **Second Pillar of OOP: Inheritance**
- Third Pillar of OOP: Polymorphism
- `System.Object`

What is Inheritance?

- Inheritance specifies an “is-a” relationship between classes



Generalization

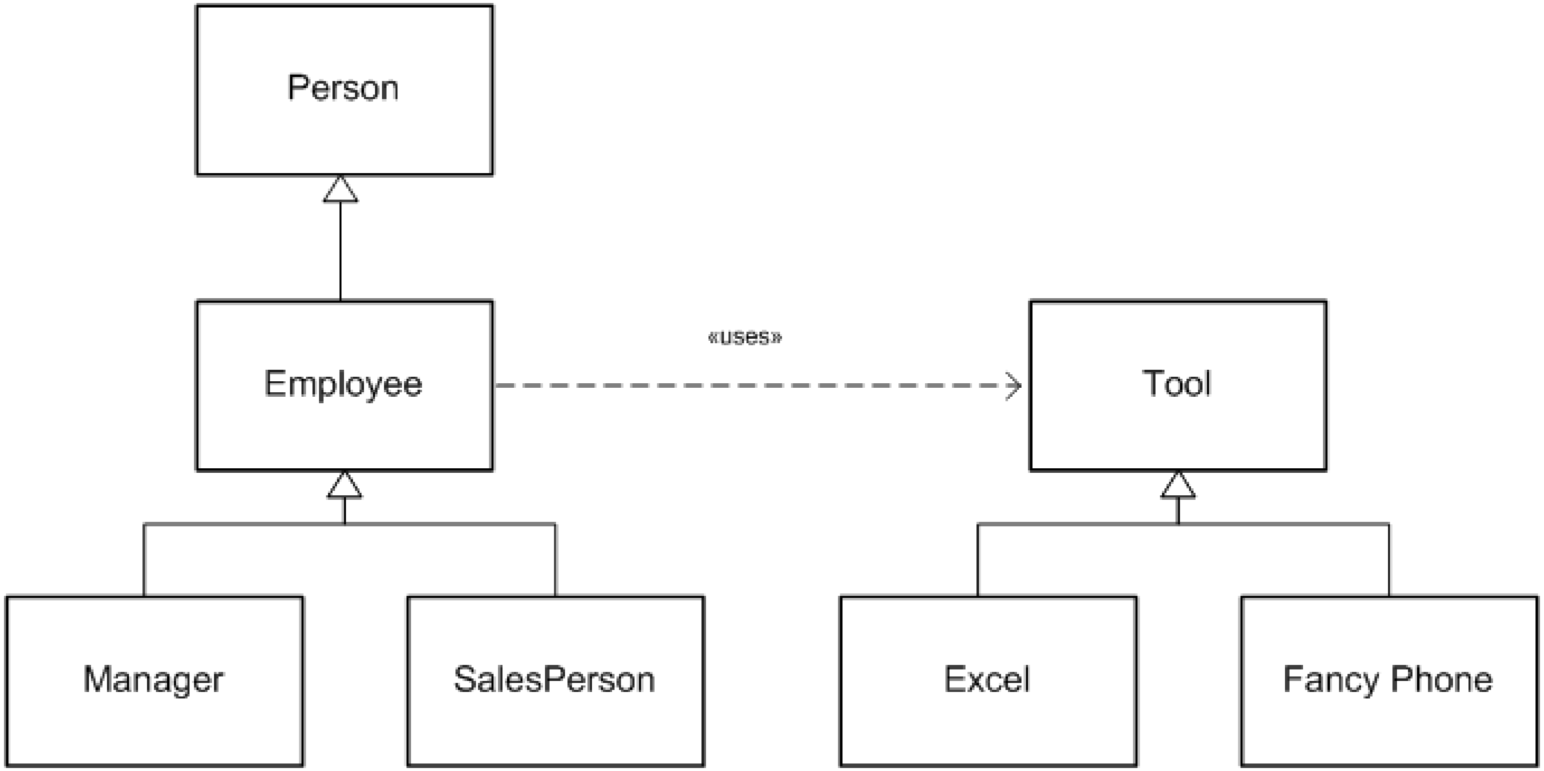
↕

Specialization

- New classes are said to *specialize* base classes
- Has all the characteristics + maybe more

- Single vs. Multiple inheritance

Class Hierarchies



Base Classes

- Create a derived class using ‘:’ in class definition

```
class Car
{
   public readonly int maxSpeed;
   private int currentSpeed;

   public Car( int maxSpeed = 100 )
   {
      this.maxSpeed = maxSpeed;
   }
}
```

```
class MiniVan : Car
{
   public MiniVan()
   {
      ...
   }
}
```

```
MiniVan van = new MiniVan();
Console.WriteLine( van.maxSpeed );      ✓
Console.WriteLine( van.currentSpeed ); ✗
```

- Inherits all public members
- Can only derive from a single base class! But…

Sealed Classes

- Classes can explicitly prevent inheritance

```
sealed class MiniVan : Car
{
   public MiniVan()
   {
      ...
   }
}
```

```
class DeluxeMiniVan : MiniVan
{
   ...
}
```

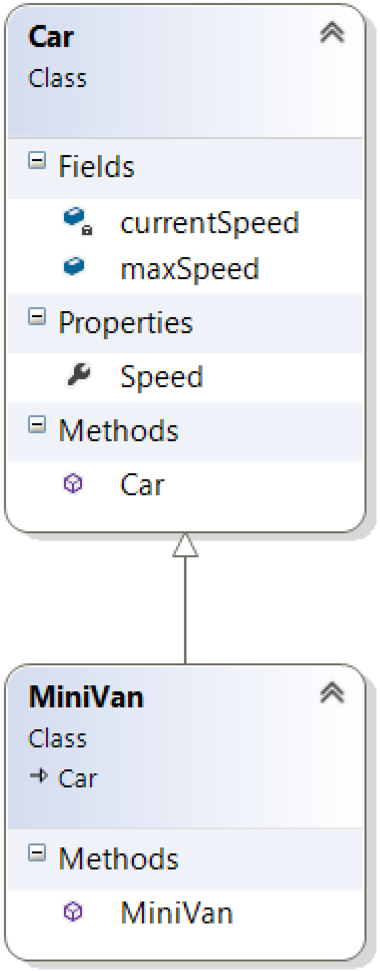
✗

- A lot of .NET Framework classes are sealed, e.g. `System.String`

Class Diagrams in Visual Studio

- Class diagrams can be easily visualized in Visual Studio

- "Add New Item" -> "Class Diagram", or
- Project node -> "View Class Diagram"



Car
Class

⊟ Fields
currentSpeed
maxSpeed

⊟ Properties
Speed

⊟ Methods
Car

MiniVan
Class
→ Car

⊟ Methods
MiniVan

The base Keyword

- The **base** keyword is used to control base class creation

```
class Car
{
   public readonly int maxSpeed;
   private int currentSpeed;

   public Car( int maxSpeed = 110 )
   {
      this.maxSpeed = maxSpeed;
   }
}
```

```
class MiniVan : Car
{
   public MiniVan() :
      base( 90 )
   {
   }
}
```

```
MiniVan van = new MiniVan();
Console.WriteLine( van.maxSpeed ); // 90
```

- This is very similar to the **this** keyword, but for base classes

The protected Modifier

- Protected members are visible to derived classes also

```
class Car
{
   public readonly int maxSpeed;
   protected int currentSpeed;

   public Car( int maxSpeed = 110
)
   {
      this.maxSpeed = maxSpeed;
   }
}
```

```
class MiniVan : Car
{
   public void CutSpeed()
   {
      currentSpeed /= 2;
   }
}
```

```
MiniVan van = new MiniVan();
van.CutSpeed();                      ✓
Console.WriteLine( van.currentSpeed ); ✗
```

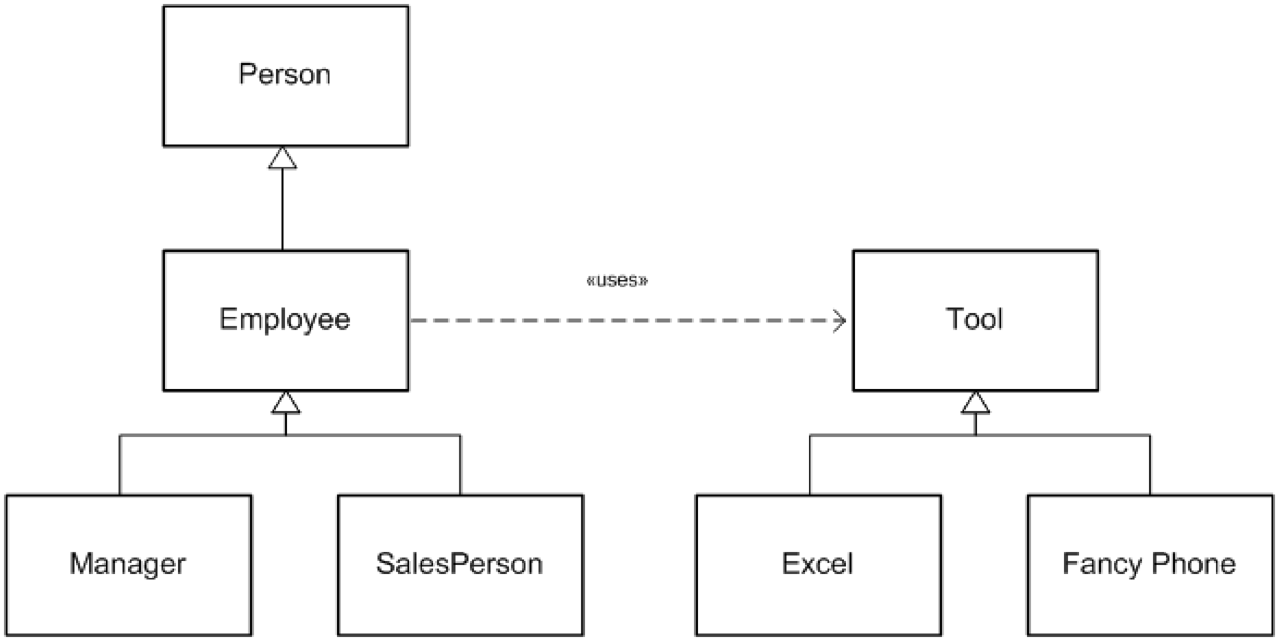
- But still not visible to the outside!

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- **Third Pillar of OOP: Polymorphism**
- `System.Object`

What is Polymorphism?

- Polymorphism
  - The ability of objects belonging to related classes to respond to method calls of methods of the same name, each one according to an appropriate type-specific behavior



Virtual Methods

- Mark *virtual methods* with the **`virtual`** keyword

```
class Employee
{
   float _currentPay;

   public virtual void GiveBonus( float amount )
   {
      _currentPay += amount;
   }
}
```

- This allows behavior to be overridden in subclasses

Overriding Virtual Methods

- Override behavior using the **override** keyword

```
class Manager : Employee
{
   public int NumberOfOptions { get; protected set; }

   public override void GiveBonus( float amount )
   {
      base.GiveBonus( amount );

      Random r = new Random();
      NumberOfOptions += r.Next( 500 );
   }
}
```

- Use the **base** keyword to leverage parent implementation

Sealing Virtual Members

- Virtual methods can be sealed to prevent overriding

```
class SalesPerson : Employee
{
   public sealed override void GiveBonus( float amount )
   {
      int salesBonus = 0;
      ...
      base.GiveBonus( amount * salesBonus );
   }
}
```

```
class FreelanceSalesman : SalesPerson
{
   public int HoursWorked { get; protected set; }

   public override void GiveBonus( float amount )
   {
      base.GiveBonus( amount + HoursWorked * 2 );
   }
}
```

Abstract Classes

- Sometimes it does not make sense to instantiate certain classes
- Such classes are *abstract* classes

```
abstract class Employee
{
   public string Name { get; protected set; }
   private float _currentPay;

   public Employee( string name, float currentPay )
   {
      Name = name;
      _currentPay = currentPay;
   }
}
```

Abstract Methods

- An *abstract method* is a requirement to derived classes to implement it

```
abstract class Shape
{
   protected string _shapeName;

   public abstract void Draw( );
}
```

```
class Hexagon : Shape
{
   public override void Draw()
   {
      ...
   }
}
```

✓

```
class Circle : Shape
{
   public Circle()
   {
   }
}
```

✗

- An abstract method is a virtual method which must be overridden
- Abstract methods must occur only in abstract classes

Member Shadowing

- The inverse of overriding is *shadowing* members
- Use the new keyword to
  - Resolve name clashes in code
  - Hide methods with identical signature

```
class FrameworkClass
{
   public void Clear() { ... }
}
```

```
class MyClass : FrameworkClass
{
   public new void Clear()
   {
   }
}
```

  - Can hide both virtual and non-virtual members
- Can be used to hide also data members

Parent/Child Conversions

- Conversion from child to parent class reference
  - Can be implicit or explicit
  - Never fails!
  - Can always be assigned to object
- Conversion from parent to child class reference
  - Has to be explicit
  - Runtime-checks the underlying type of object
  - Will throw an `InvalidCastException` if conversion is illegal

The `is` Operator

- The **`is`** operator checks whether a conversion can be made

```
Employee e = new Manager( ... );
...
if( e is Manager )
{
   Manager m = (Manager) e;

   Console.WriteLine( m.NumberOfOptions );
}
```

The as Operator

- The **as** operator performs conversion if it can be made
  - Otherwise null is returned
  - Exceptions are never thrown!

```
Employee e = new Manager( ... );
...
Manager m = e as Manager;
if( m != null )
{
   Console.WriteLine( m.NumberOfOptions );
}
```

Agenda

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- **`System.Object`**

`System.Object` Members

- Every class ultimately derives from `System.Object`
- This master parent class is captured by the `object` keyword

Name	Characteristics
`ToString()`	Virtual
`Equals()`	Virtual
`GetHashCode()`	Virtual
`Finalize()`	Virtual
`GetType()`	Non-virtual
`MemberwiseClone()`	Non-virtual
`Equals()`	Static
`ReferenceEquals()`	Static

Overriding ToString()

- Override the **ToString()** method to provide a string representation for the object

```
abstract class Employee
{
    ...
    public override string ToString()
    {
        return string.Format( "Employee named \"{0}\"", Name );
    }
}
```

```
Manager manager = new Manager( "Angry Bob", ... );

Console.WriteLine( manager ); // ???
```

Overriding Equals()

- Override the **`Equals()`** method to provide custom equality

```
abstract class Employee
{  ...
   public override bool Equals( object obj )
   {  ...
      if (other.Name == this.Name )
      {
         return true;
      }
      ...
      return false;
   }
}
```

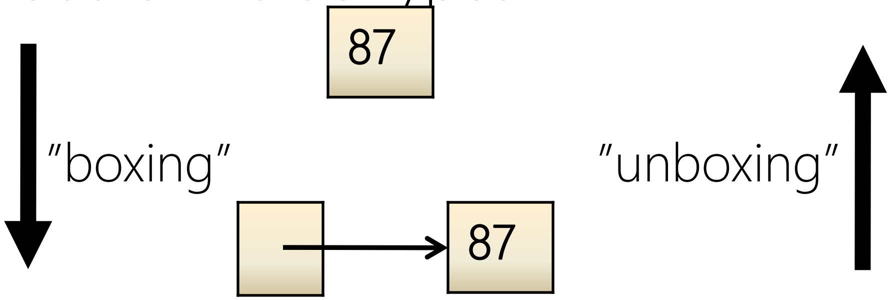
```
Manager m1 = new Manager(
   "Angry Bob", 900000, 1000 );
Manager m2 = new Manager(
   "Angry Bob", 900000, 1000 );

Console.WriteLine( m1.Equals( m2 ) );
Console.WriteLine( m1 == m2 );
```

- Does not influence the **`==`** operator!

Boxing and Unboxing

- Value types can be boxed as reference types
- This unified type system has many advantages, e.g. calling object methods on value types

```
int i = 87;
object o = i;
```



```
int i = (int) o;
```

- Downside is performance and safety
  - Can raise **`InvalidCastException`**

Quiz: Inheritance and Polymorphism – Right or Wrong?

✓
```
abstract class Employee {
   public string Name
      { get; protected set; }
   public abstract bool Work();
}
```

✓
```
class Developer : Employee {
   public override bool Work()
   {
      Name = "Hard-worker!";
      return true;
   }
}
```

✗
```
class Manager : Employee
{
   public bool Work()
   { return false; }
}
```

✗
```
Employee e = new Employee();
```

✓
```
Employee e = new Developer();
```

✓
```
Developer d = new Developer();
```

✗
```
Developer d = new Employee();
```

✗
```
Developer d = new Manager();
```

✓
```
Developer d = new Developer();
Console.WriteLine( d.Name );
```

✗
```
Developer d = new Developer();
d.Name = "Geek!";
```

✓
```
Employee e1 = new Developer();
Employee e2 = new Manager();
Console.WriteLine(
   e1.Work() == e2.Work()
);
```

Summary

- Introducing Object-Oriented Programming
- First Pillar of OOP: Encapsulation
- Creating Classes and Objects
- Access Modifiers
- Static Classes and Members
- Properties and Initializers
- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`

Question 1

You have written the following program.

The Hobby property must be accessed only by code deriving from the Person class. The Hobby property is allowed to be modified by code in the Person class only.

```
01 public class Person
02 {
03     internal string Hobby
04     {
05         get;
06         set;
07     }
08 }
```

Which two actions should you perform?
(Each correct answer constitutes part of the complete solution)

a) Replace line 03 with: **`public string Hobby`**

b) [✓ Replace line 03 with: **`protected string Hobby`**]

c) Replace line 05 with: **`protected get;`**

d) Replace line 05 with: **`private get;`**

e) [✓ Replace line 06 with: **`private set;`**]

f) Replace line 06 with: **`protected set;`**

Question 2

You have written the following class Vehicle. You must ensure that no subclass of Vehicle overrides the ToString() method.

```
01 abstract public class Vehicle
02 {
03     public override string ToString()
04     {
05         ...
06     }
07 }
```

Which action should you perform?

a) Add the **`sealed`** keyword to line 01

[✓ b) Add the **`sealed`** keyword to line 03]

c) Add the **`abstract`** keyword to line 03

d) Add the **`new virtual`** keyword to line 03