Module 8

"Inheritance and Polymorphism"

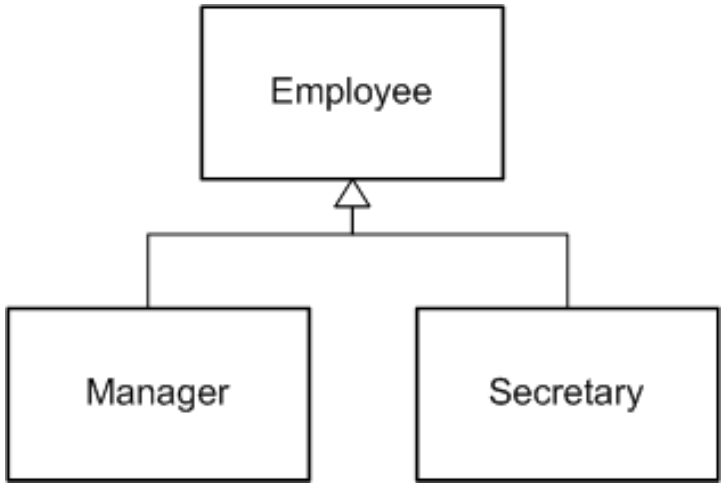
TEKNOLOGISK
INSTITUT

Agenda

- **Second Pillar of OOP: Inheritance**
- Third Pillar of OOP: Polymorphism
- Pattern Matching Types
- `System.Object`
- Lab 8
- Discussion and Review

What is Inheritance?

- Inheritance specifies an "is-a" relationship between classes



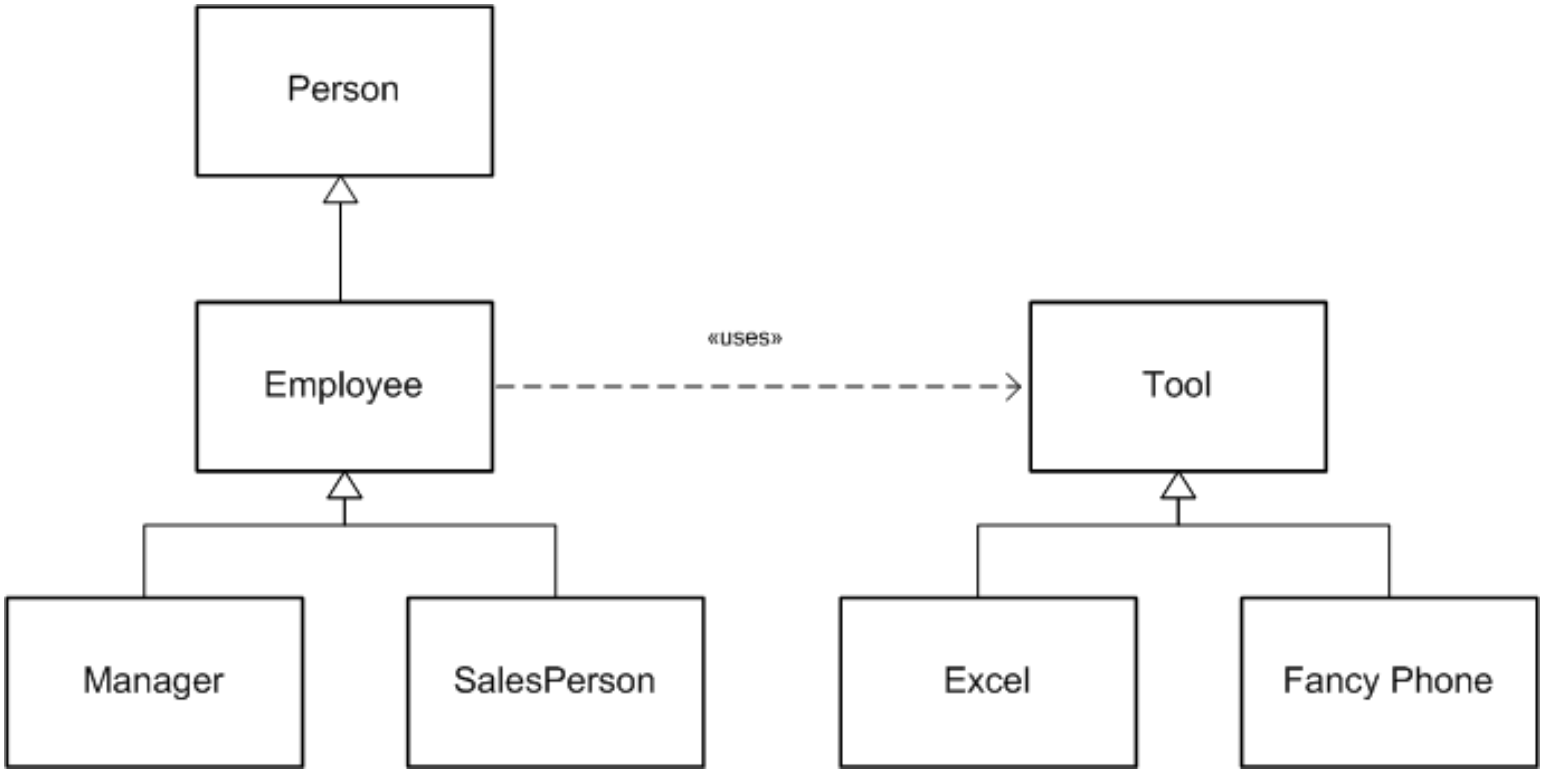
Generalization

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()
   {
      ...
   }
}
```

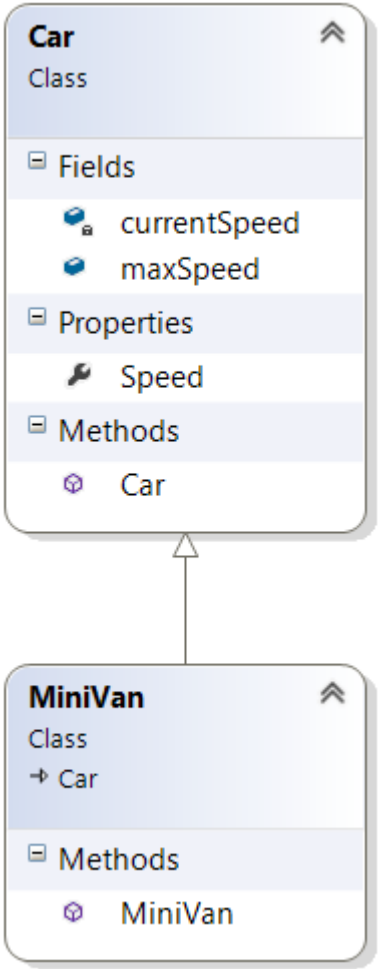
```
class DeluxeMiniVan : MiniVan
{
   ...
}
```

- 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"



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

TEKNOLOGISK INSTITUT

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 );
```

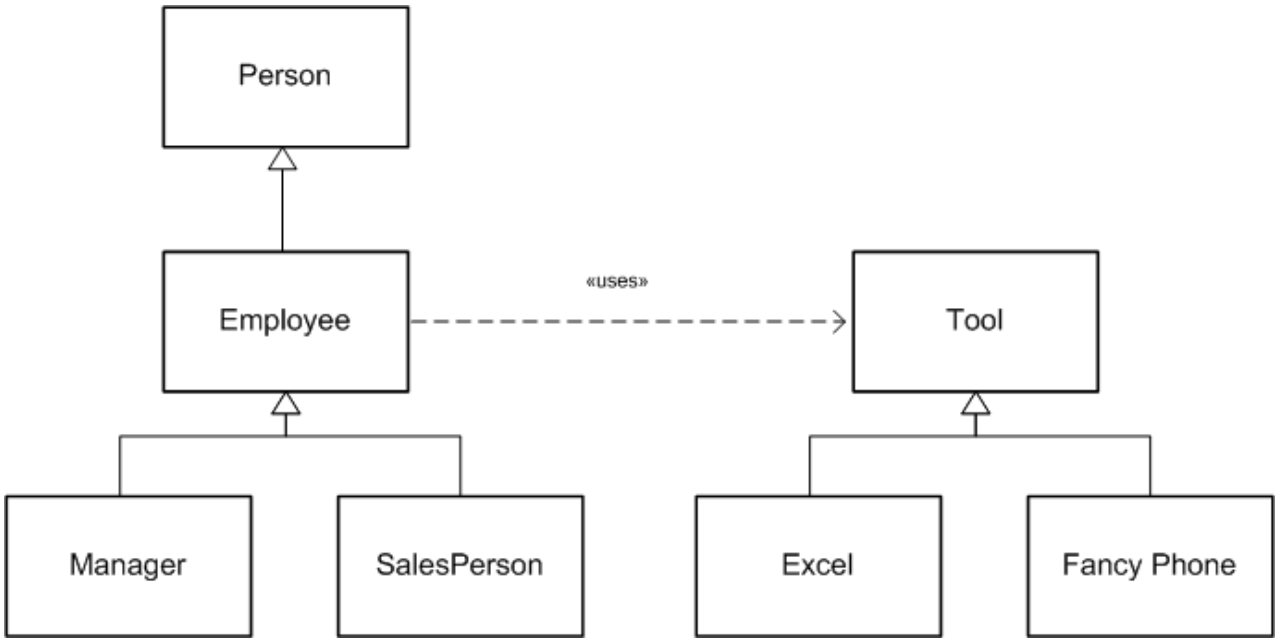
- But still not visible to the outside!

Agenda

- Second Pillar of OOP: Inheritance
- **Third Pillar of OOP: Polymorphism**
- Pattern Matching Types
- **`System.Object`**
- Lab 8
- Discussion and Review

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

TEKNOLOGISK INSTITUT

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

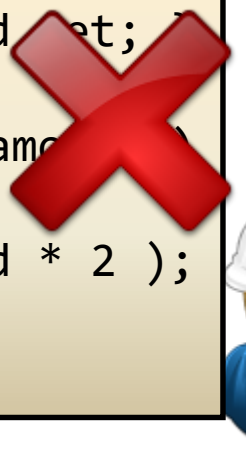
Sealing Virtual Members

- 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;
   }
}
```

TEKNOLOGISK INSTITUT

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

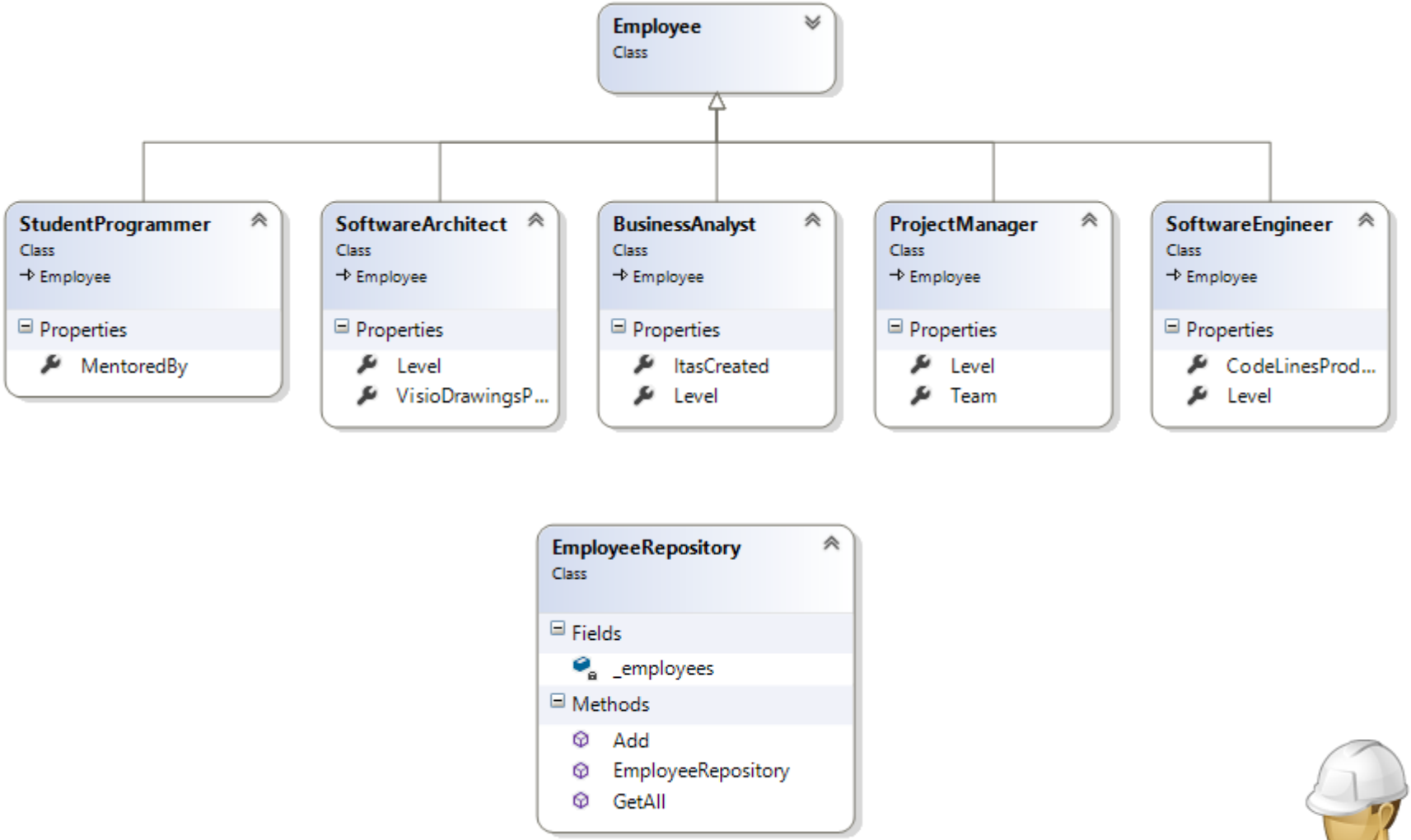
- The **as** operator performs conversion if it can be made
  - Otherwise null is returned
  - Exceptions are never thrown!

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

Agenda

- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- **Pattern Matching Types**
- **`System.Object`**
- Lab 8
- Discussion and Review

Example: Employee



Pattern Matching with is

- Three types of patterns for matching in C# 7
  - Constant patterns *c* e.g. **`null`**
  - Type patterns *T x* e.g. **`int x`**
  - Var patterns **`var`** *x*
- Matches and/or captures to identifiers to nearest surrounding scope

```
foreach (Employee e in all)
{
    if (e is SoftwareEngineer se)
    {
        Console.WriteLine($"{se.FullName} has produced" +
                          $"{se.CodeLinesProduced} lines of C#");
    }
}
```

- The **`is`** keyword is now compatible with patterns

Type Switch with Pattern Matching

- Can switch on any type
  - Case clauses can make use of patterns and new **when** conditions

```
Employee e = ...;
switch (e)
{
    case SoftwareArchitect sa:
        WriteLine($"{sa.FullName} plays with Visio");
        break;
    case SoftwareEngineer se when
         se.Level == SoftwareEngineerLevel.Lead:
        WriteLine($"{se.FullName} is a lead software engineer");
        break;
    case null:
    default:
        break;
}
```

Agenda

- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- Pattern Matching Types
- **`System.Object`**
- Lab 8
- Discussion and Review

`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;
   }
}
```

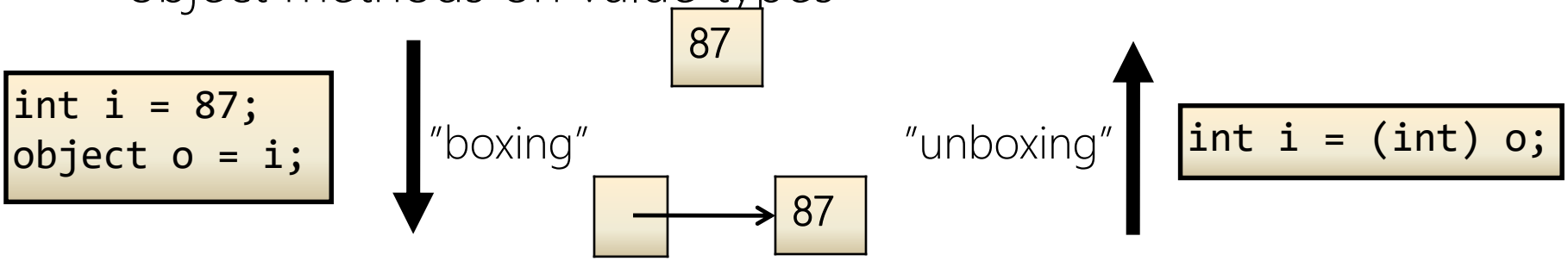
```
Manager m1 = new Manager(
   "Angry Bob", 900000, 1000 );
Manager m2 = new Manager(
   "Angry Bob", 900000, 1000 );

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



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

Quiz: Inheritance and Polymorphism – Right or Wrong?

TEKNOLOGISK INSTITUT

```
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()
);
```

TEKNOLOGISK
INSTITUT

Lab 8: Using Inheritance and Polymorphism

Discussion and Review

- Second Pillar of OOP: Inheritance
- Third Pillar of OOP: Polymorphism
- `System.Object`


WINCUBATE
Jesper Gulmann Henriksen
PhD, MCT, MCSD, MCPD
Phone : +45 22 12 36 31
Email : jgh@wincubate.net
WWW : http://www.wincubate.net
Hasselvangen 243
8355 Solbjerg
Denmark