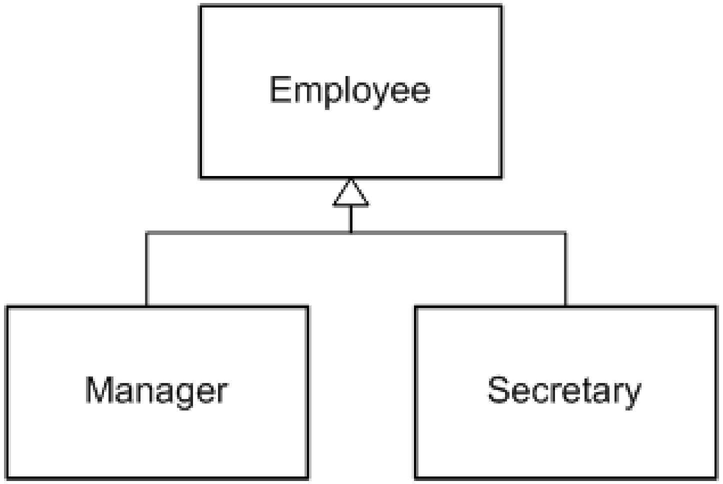
TEKNOLOGISK
INSTITUT

Module 8

Inheritance and Polymorphism

What is Inheritance?

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



Generalization

↕

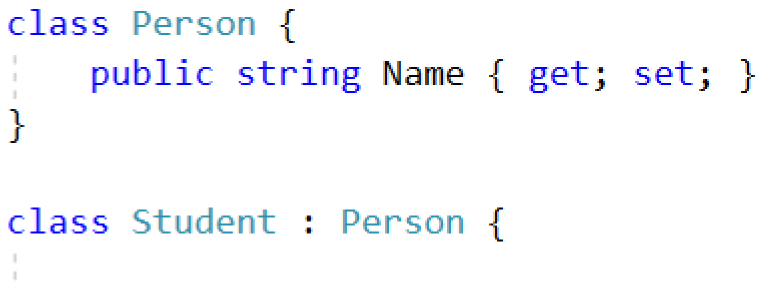
Specialization

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

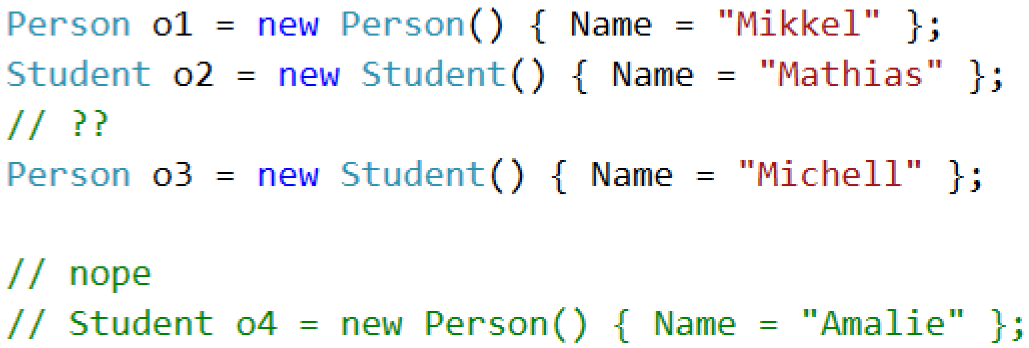
- Single vs. Multiple inheritance

Base Classes

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



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

class Student : Person {

}
```

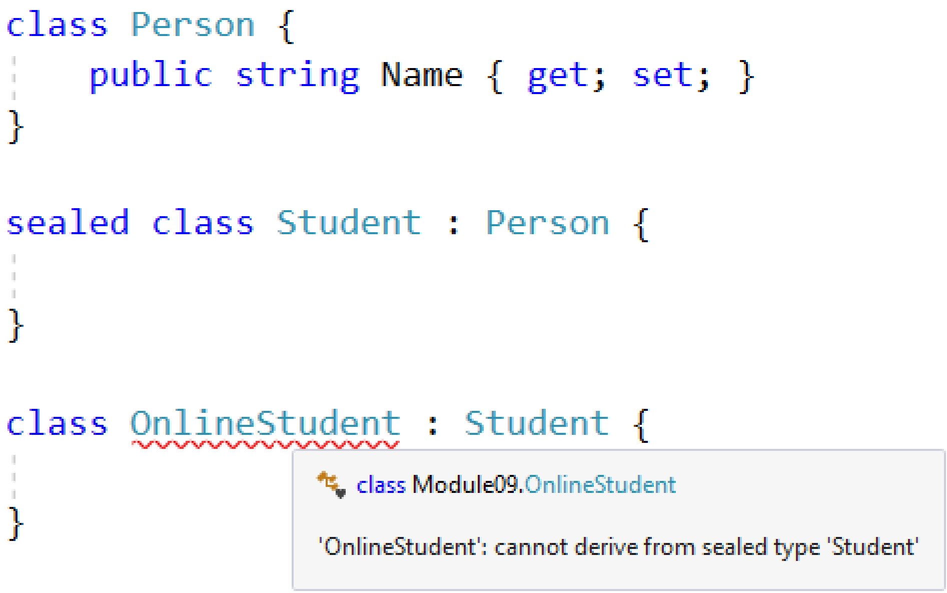


```
Person o1 = new Person() { Name = "Mikkel" };
Student o2 = new Student() { Name = "Mathias" };
// ??
Person o3 = new Student() { Name = "Michell" };

// nope
// Student o4 = new Person() { Name = "Amalie" };
```

- Inherits all public membe
  - Not constructors
- Can only derive from a single base class!

Sealed Classes

- Classes can explicitly prevent inheritance



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

sealed class Student : Person {

}

class OnlineStudent : Student {

}
```

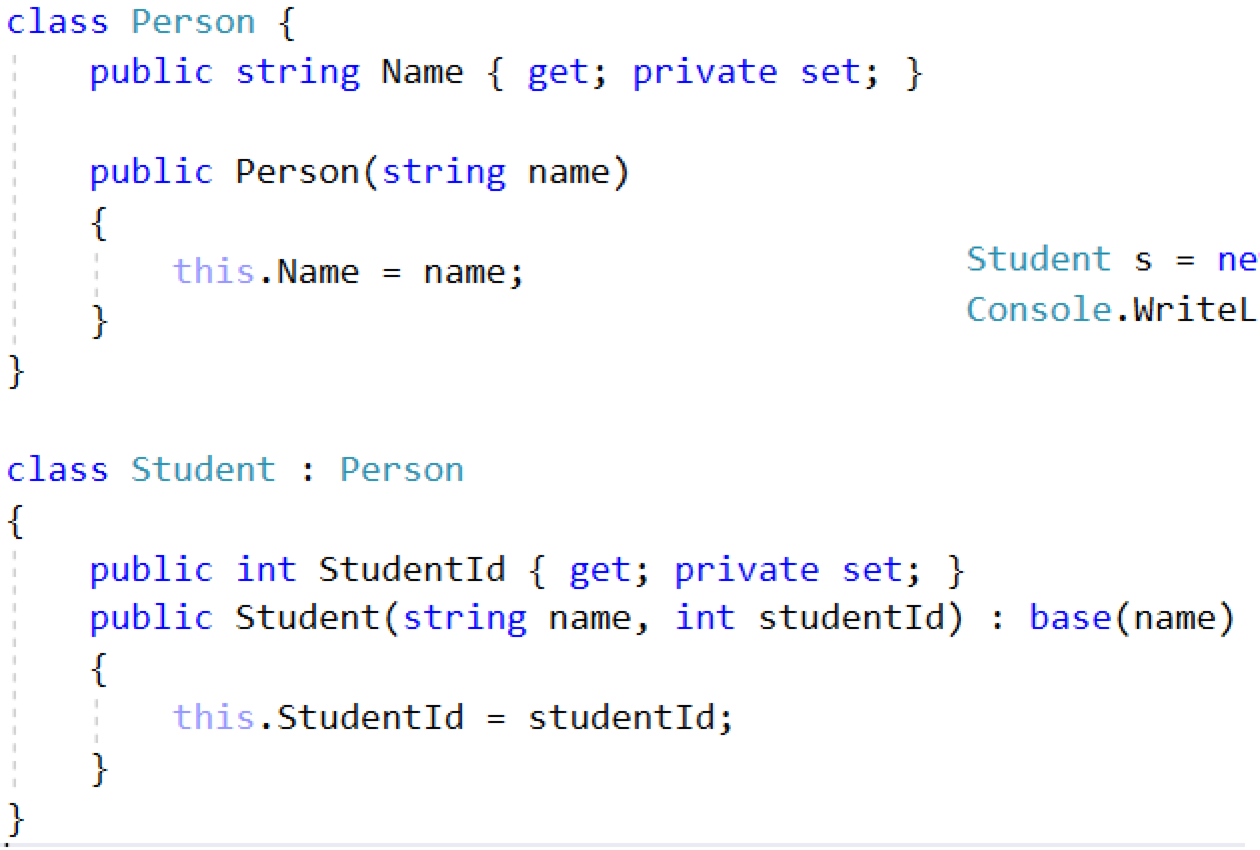
class Module09.OnlineStudent

'OnlineStudent': cannot derive from sealed type 'Student'

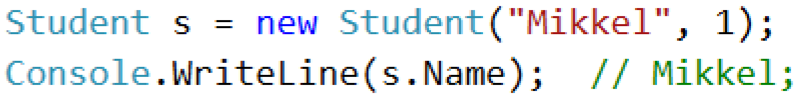
- A lot of .NET Framework classes are sealed, e.g. `System.String`

The base Keyword

- The base keyword is used to control base class creation



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

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

class Student : Person
{
    public int StudentId { get; private set; }
    public Student(string name, int studentId) : base(name)
    {
        this.StudentId = studentId;
    }
}
```

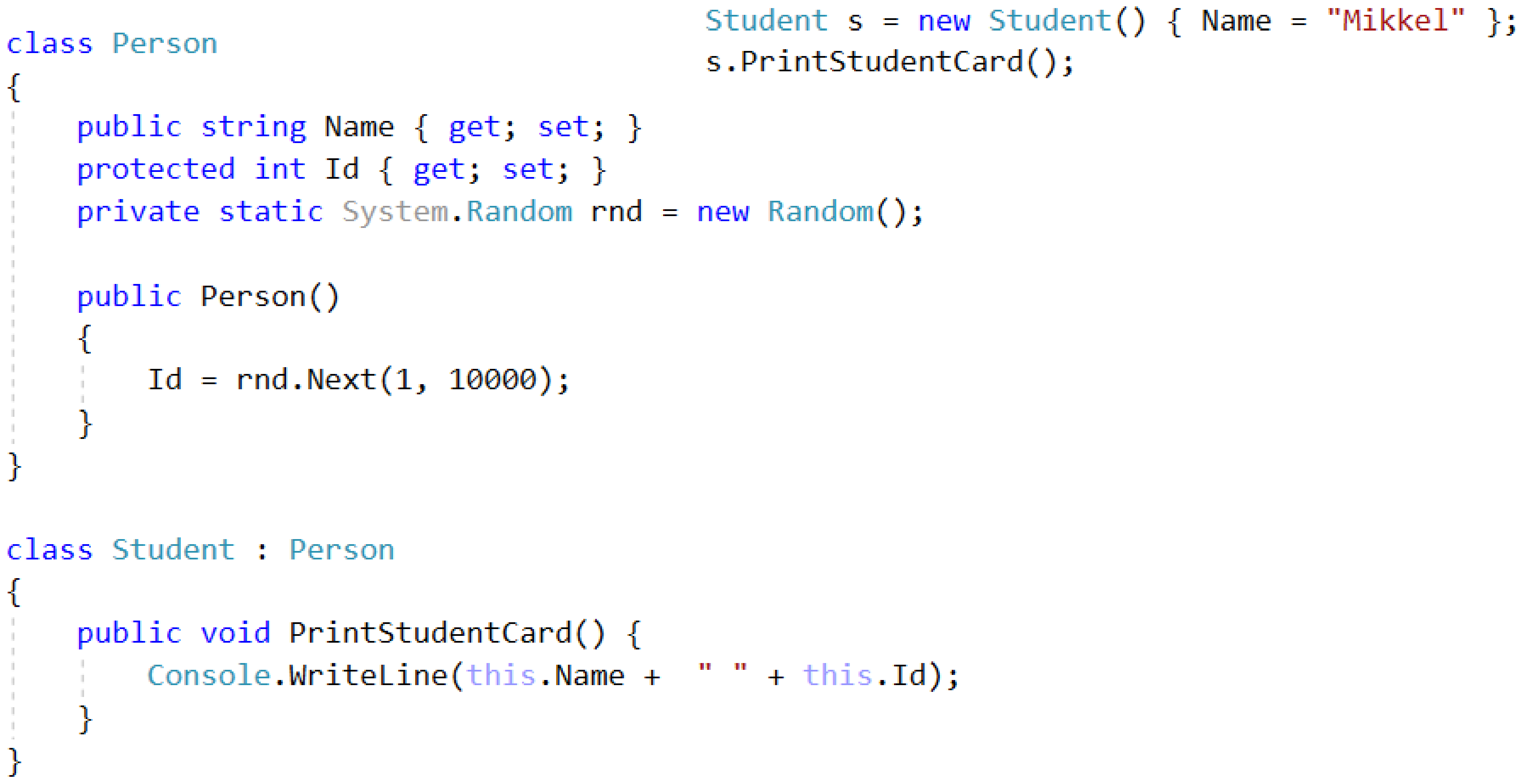


```
Student s = new Student("Mikkel", 1);
Console.WriteLine(s.Name);  // Mikkel;
```

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

The `protected` Modifier

- Protected members are visible to derived classes also
- But still not visible to the outside!

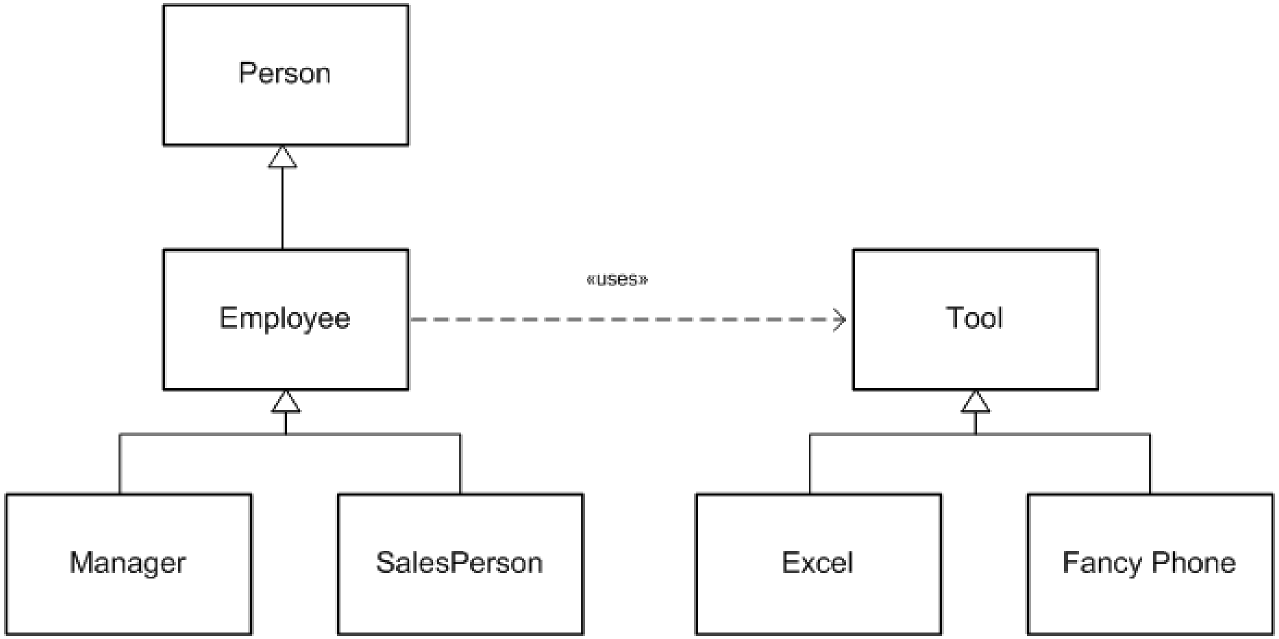


```
Student s = new Student() { Name = "Mikkel" };
s.PrintStudentCard();
```

```
class Person
{
    public string Name { get; set; }
    protected int Id { get; set; }
    private static System.Random rnd = new Random();

    public Person()
    {
        Id = rnd.Next(1, 10000);
    }
}

class Student : Person
{
    public void PrintStudentCard() {
        Console.WriteLine(this.Name +  " " + this.Id);
    }
}
```

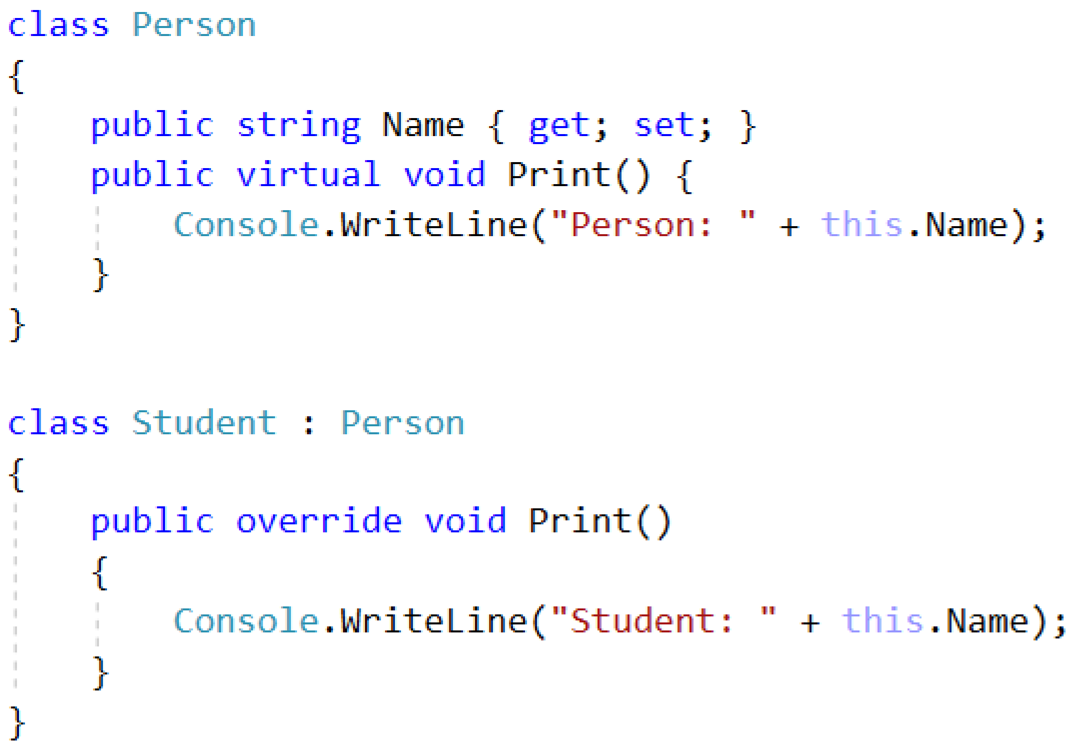
What is Polymorphism?

- 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

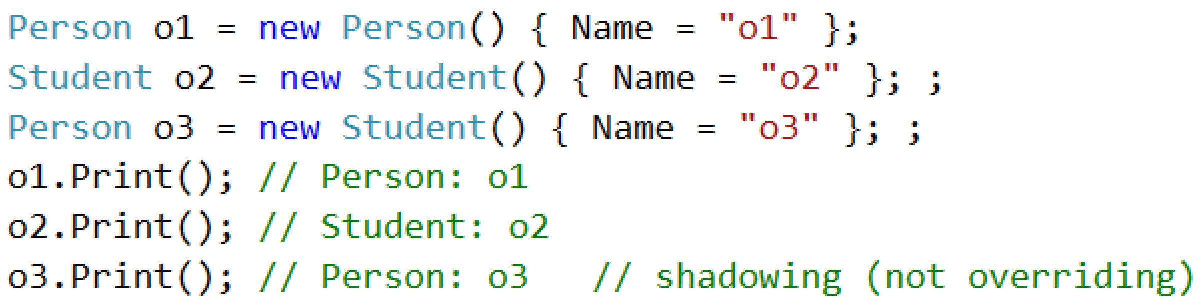
- Mark *virtual methods* with the `virtual` keyword
- This allows behavior to be overridden in subclasses
- Reuse with the Base keyword
- Override with the override keyword



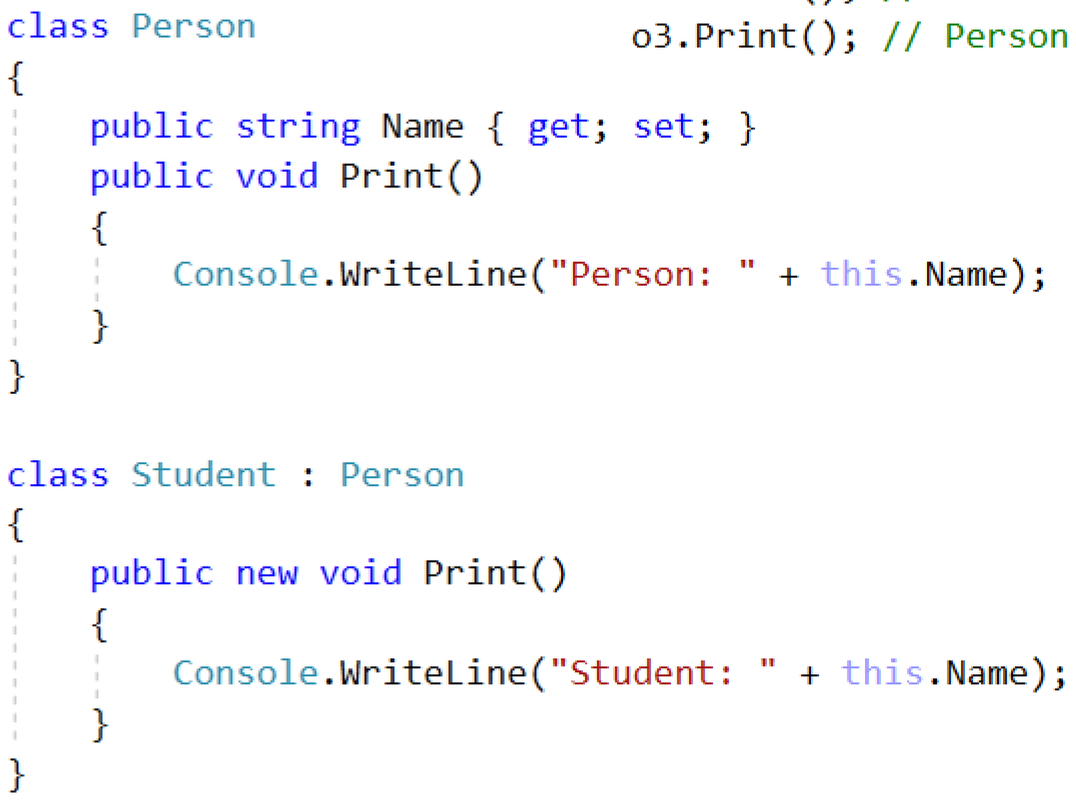
```
class Person
{
    public string Name { get; set; }
    public virtual void Print() {
        Console.WriteLine("Person: " + this.Name);
    }
}

class Student : Person
{
    public override void Print()
    {
        Console.WriteLine("Student: " + this.Name);
    }
}
```

Member Shadowing

- The inverse of overriding is *shadowing* members
- Use the new keyword



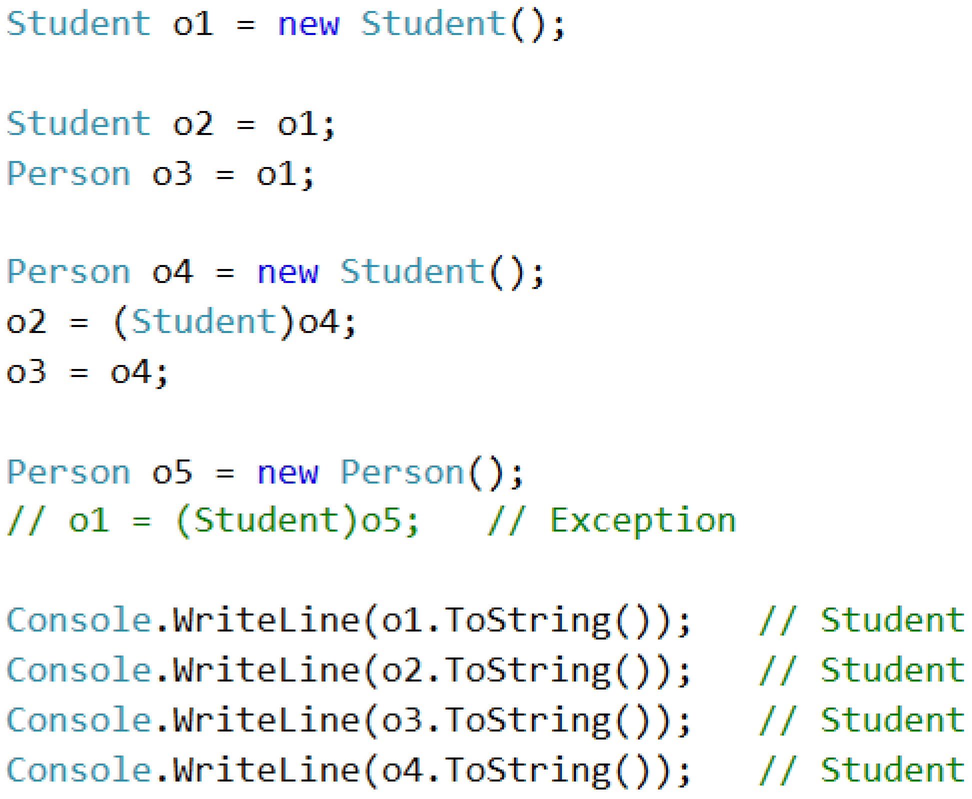
```
Person o1 = new Person() { Name = "o1" };
Student o2 = new Student() { Name = "o2" }; ;
Person o3 = new Student() { Name = "o3" }; ;
o1.Print(); // Person: o1
o2.Print(); // Student: o2
o3.Print(); // Person: o3   // shadowing (not overriding)
```



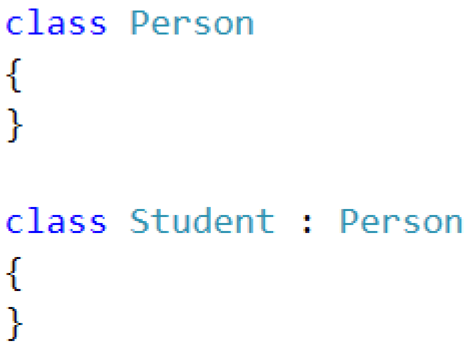
```
class Person
{
    public string Name { get; set; }
    public void Print()
    {
        Console.WriteLine("Person: " + this.Name);
    }
}

class Student : Person
{
    public new void Print()
    {
        Console.WriteLine("Student: " + this.Name);
    }
}
```

Child Conversions

- Conversion from child to parent class reference
  - Can be implicit or explicit
  - Never fails!



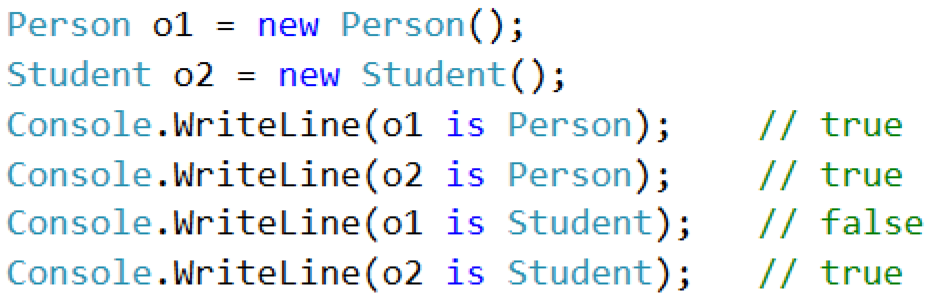
```
Student o1 = new Student();

Student o2 = o1;
Person o3 = o1;

Person o4 = new Student();
o2 = (Student)o4;
o3 = o4;

Person o5 = new Person();
// o1 = (Student)o5;    // Exception

Console.WriteLine(o1.ToString());    // Student
Console.WriteLine(o2.ToString());    // Student
Console.WriteLine(o3.ToString());    // Student
Console.WriteLine(o4.ToString());    // Student
```



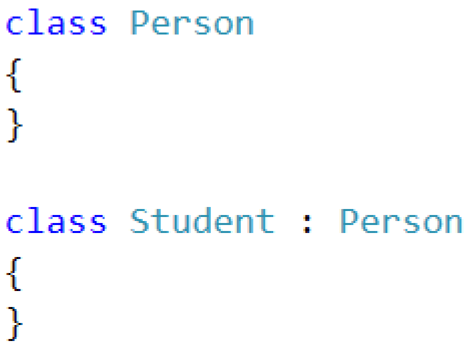
```
class Person
{
}

class Student : Person
{
}
```

The `is` Operator

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



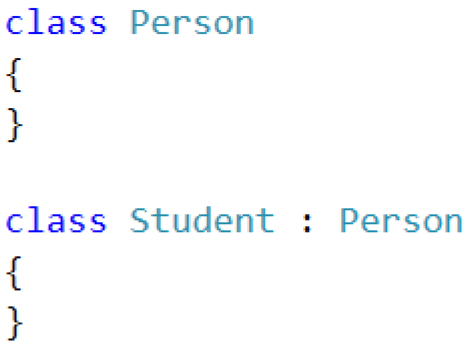
```
Person o1 = new Person();
Student o2 = new Student();
Console.WriteLine(o1 is Person);    // true
Console.WriteLine(o2 is Person);    // true
Console.WriteLine(o1 is Student);   // false
Console.WriteLine(o2 is Student);   // true
```



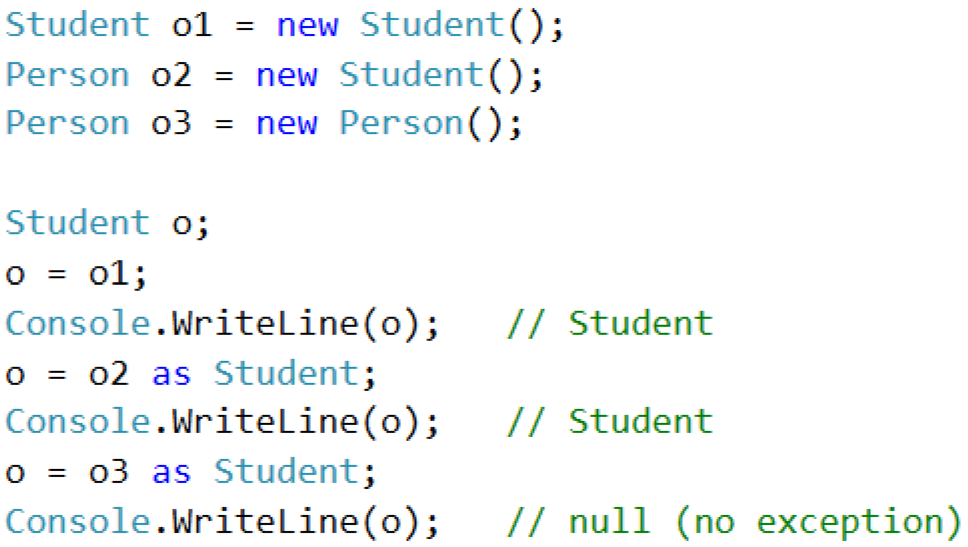
```
class Person
{
}

class Student : Person
{
}
```

The as Operator

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



```
class Person
{
}

class Student : Person
{
}
```



```
Student o1 = new Student();
Person o2 = new Student();
Person o3 = new Person();

Student o;
o = o1;
Console.WriteLine(o);   // Student
o = o2 as Student;
Console.WriteLine(o);   // Student
o = o3 as Student;
Console.WriteLine(o);   // null (no exception)
```

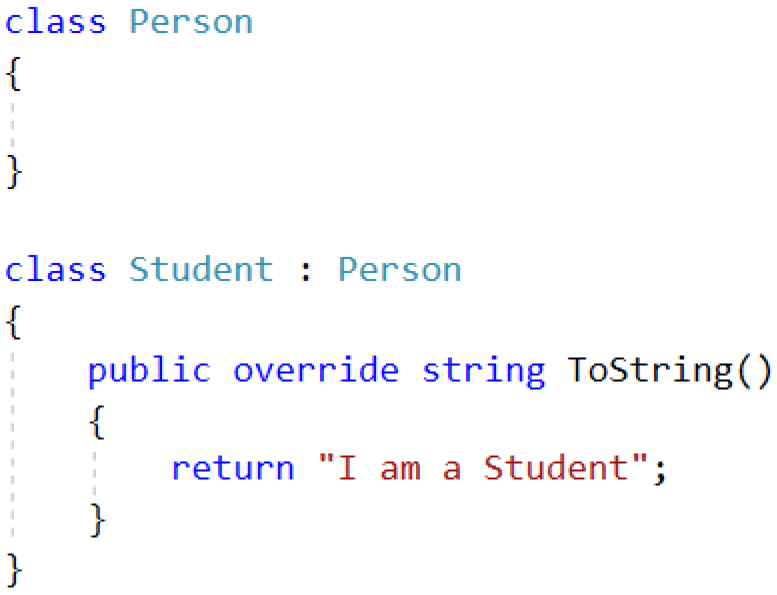
`System.Object` Members

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

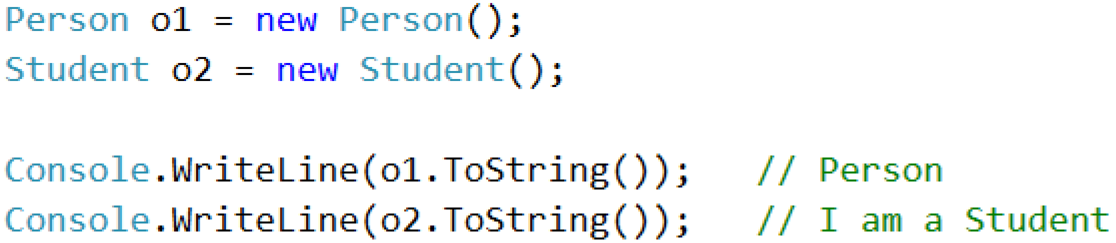
Name	Characteristics
`ToString()`	Virtual
`Equals()`	Virtual
`GetHashCode()`	Virtual
`Finalize()`	Virtual
`GetType()`	Non-virtual
`MemberwiseClone()`	Non-virtual
`Equals()`	Static
`ReferenceEquals()`	Static

Overriding ToString()

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



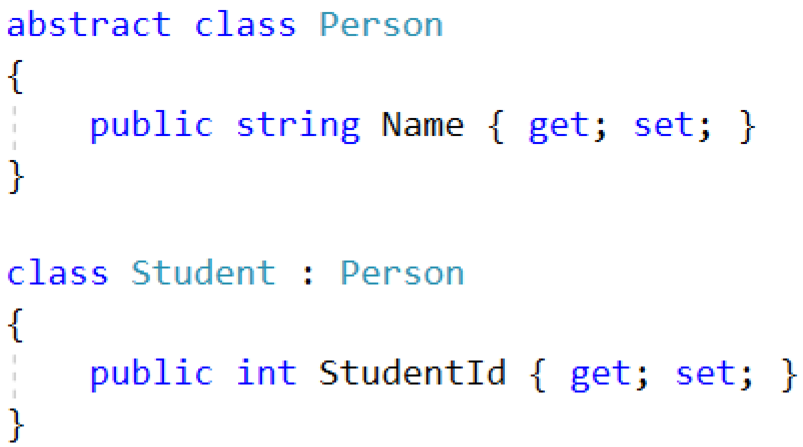
```
class Person
{

}

class Student : Person
{
    public override string ToString()
    {
        return "I am a Student";
    }
}
```



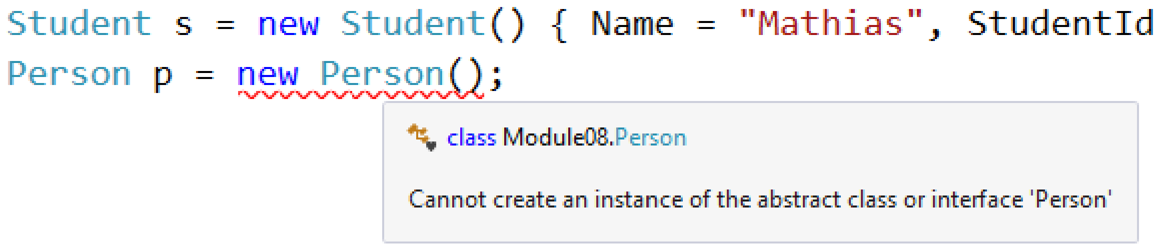
```
Person o1 = new Person();
Student o2 = new Student();

Console.WriteLine(o1.ToString());   // Person
Console.WriteLine(o2.ToString());   // I am a Student
```

Abstract Classes

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



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

class Student : Person
{
    public int StudentId { get; set; }
}
```



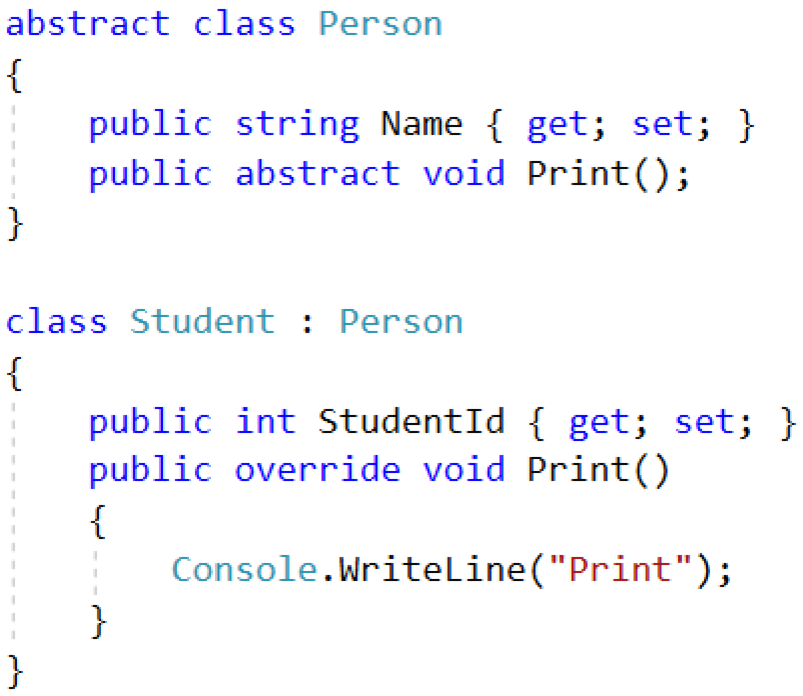
```
Student s = new Student() { Name = "Mathias", StudentId = 1 };
Person p = new Person();
```

class Module08.Person

Cannot create an instance of the abstract class or interface 'Person'

Abstract Methods

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



```
abstract class Person
{
    public string Name { get; set; }
    public abstract void Print();
}

class Student : Person
{
    public int StudentId { get; set; }
    public override void Print()
    {
        Console.WriteLine("Print");
    }
}
```

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